Application-Transparent Fault Management

Mark E. Russinovich

Research Report No. CMUCSC-94-8

August 1994

19951012 040

~~19950505 104~~

OFFICE OF THE UNDER SECRETARY OF DEFENSE (ACQUISITION)
DEFENSE TECHNICAL INFORMATION CENTER
CAMERON STATION
ALEXANDRIA, VIRGINIA 22304-6145

IN REPLY REFER TO DTIC-OCC

SUBJECT: Distribution Statements on Technical Documents

TO: OFFICE OF NAVAL RESEARCH
CORPORATE PROGRAMS DIVISION
ONR 353
800 NORTH QUINCY STREET
ARLINGTON, VA 22217-5660

19951012 040

1. Reference: DoD Directive 5230.24, Distribution Statements on Technical Documents, 18 Mar 87.

2. The Defense Technical Information Center received the enclosed report (referenced below) which is not marked in accordance with the above reference.

TECH REPORT
N00014-91-J-4139
TITLE: FAULT-TOLERANT MACH

3. We request the appropriate distribution statement be assigned and the report returned to DTIC within 5 working days.

4. Approved distribution statements are listed on the reverse of this letter. If you have any questions regarding these statements, call DTIC's Cataloging Branch, (703) 274-6837.

FOR THE ADMINISTRATOR:

1 Encl

GOPALAKRISHNAN NAIR
Chief, Cataloging Branch

FL-171
Jul 93

DISTRIBUTION STATEMENT A:

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION IS UNLIMITED

DISTRIBUTION STATEMENT B:

DISTRIBUTION AUTHORIZED TO U.S. GOVERNMENT AGENCIES ONLY; (Indicate Reason and Date Below). OTHER REQUESTS FOR THIS DOCUMENT SHALL BE REFERRED TO (Indicate Controlling DoD Office Below).

DISTRIBUTION STATEMENT C:

DISTRIBUTION AUTHORIZED TO U.S. GOVERNMENT AGENCIES AND THEIR CONTRACTORS; (Indicate Reason and Date Below). OTHER REQUESTS FOR THIS DOCUMENT SHALL BE REFERRED TO (Indicate Controlling DoD Office Below).

DISTRIBUTION STATEMENT D:

DISTRIBUTION AUTHORIZED TO DOD AND U.S. DOD CONTRACTORS ONLY; (Indicate Reason and Date Below). OTHER REQUESTS SHALL BE REFERRED TO (Indicate Controlling DoD Office Below).

DISTRIBUTION STATEMENT E:

DISTRIBUTION AUTHORIZED TO DOD COMPONENTS ONLY; (Indicate Reason and Date Below). OTHER REQUESTS SHALL BE REFERRED TO (Indicate Controlling DoD Office Below).

DISTRIBUTION STATEMENT F:

FURTHER DISSEMINATION ONLY AS DIRECTED BY (Indicate Controlling DoD Office and Date Below) or HIGHER DOD AUTHORITY.

DISTRIBUTION STATEMENT X:

DISTRIBUTION AUTHORIZED TO U.S. GOVERNMENT AGENCIES AND PRIVATE INDIVIDUALS OR ENTERPRISES ELIGIBLE TO OBTAIN EXPORT-CONTROLLED TECHNICAL DATA IN ACCORDANCE WITH DOD DIRECTIVE 5230.25, WITHHOLDING OF UNCLASSIFIED TECHNICAL DATA FROM PUBLIC DISCLOSURE, 6 Nov 1984 (Indicate date of determination). CONTROLLING DOD OFFICE IS (Indicate Controlling DoD Office).

The cited documents has been reviewed by competent authority and the following distribution statement is hereby authorized.

A
(Statement)

OFFICE OF NAVAL RESEARCH
CORPORATE PROGRAMS DIVISION
ONR 353
800 NORTH QUINCY STREET
ARLINGTON, VA 22217-5660

(Controlling DoD Office Name)

(Reason)

(Controlling DoD Office Address, City, State, Zip)

[illegible]
DEBRA T. HUGHES
DEPUTY DIRECTOR
CORPORATE PROGRAMS OFFICE

(Signature & Typed Name)

(Assigning Office)

19 SEP 1995

(Date Statement Assigned)

Abstract

As computers continue to proliferate and they are used in more demanding environments, data integrity and continuous availability are an increasingly important aspect of their designs. Since operating systems are common to all computers and it is at the operating system level where there is maximum system visibility and control, it is appropriate for the operating system to provide policies which detect, contain and tolerate faults. These policies and the mechanisms that support them form an operating system's "fault management." A fault management mechanism, the *sentry mechanism*, has been designed and implemented for a UNIX 4.3 BSD server running on the Mach 3.0 microkernel. Fault tolerant policies have been designed for a range of computer systems, from a single computer, to mirrored computers to distributed systems. The policies first addressed provide single computer applications with application-transparent fault tolerance with respect to transient faults and certain types of permanent faults. Contributions to this area include algorithms for concurrent process journaling, disk checkpointing and memory checkpointing. Formal proofs are made of the journal sequencing algorithm and the disk checkpointing algorithm. Performance measurements from an implementation of the single computer algorithms show an average performance overhead of less than 5% and a requirement of only 10 MB of dedicated disk stable storage. The system provides fault tolerance with no additional hardware other than a hard disk, and works with unmodified applications such as the X-window system. Sentry policies that provide software based fault tolerance for duplicated and triplicated computer systems as well as distributed systems have also been designed. Contributions related to these policies include mirrored system synchronization, fault detection and integration algorithms. In addition, a new n-fault tolerant distributed recovery algorithm is presented that is based on loosely synchronized checkpointing. The algorithm journals message order information, instead of using the message content based journaling of existing algorithms. Only saving information on the order of messages can potentially result in lower space and time overheads. Two variants of the algorithm are presented and formally proven. In all three system designs the sentry mechanism provides sufficient control for the fault tolerant policies.

Table of Contents

List of Figures

List of Tables

Table of Listings

Chapter 1

Background and Overview

1.1 Introduction

This thesis introduces the notion of fault management to encompass the collection of operating system policies and mechanisms employed for fault detection and recovery. The idea is advanced that similar to resource management, the operating system should provide support for other system manifestations such as detection of, and recovery from hardware faults and software failures. Fault management is already, at least implicitly, part of any computer system. In its simplest form it consists of checking system call parameters to insure they fall within valid ranges. Fault tolerance and advanced fault detection, however, has been left to highly specialized systems. As computers proliferate, an increasing number of people have come to rely on the fault-free behavior of systems they use. It has become imperative that advanced fault management be introduced into the general computing environment.

There are two distinct approaches to fault management: application dependent and application-transparent. The essential difference between the two types is that application dependency implies that the application participates actively in its own fault management while application-transparency means that the application is totally unaware of the fault management being performed on its behalf. An example of an application dependent fault management policy is a library that is linked with an application that provides the application with primitives for fault tolerant communications services. To use the library's primitives, the application must be recompiled or rewritten. An application-transparent fault management policy that provides fault tolerant services either replaces or augments the services that an application already uses. Applications therefore require no modification to use the fault management policy.

There has been a long standing division amongst professionals as to which type of fault management is more advantageous. Application-transparency is attractive because, as mentioned before, application's do not need to be changed to take advantage of fault management policies. Proponents of application dependent fault management make the case that application-transparent policies must be overly general because they cannot be fully aware of what parts of a computation are important, and that application dependency can provide much more efficient, tailored solutions to an application's needs.

This thesis does not set out to argue that one approach is better than the other. One approach does not necessarily exclude the other, and in fact, they can be complementary in nature, each addressing different fault management requirements. In the real world of computing, however, it can be argued that effective application-transparent policies are more desirable because existing solutions to problems do not have to be changed. This addresses the important issue of backward compatibility as well as the expense and time of developing customized software that uses application dependent techniques. The development of practical application-transparent policies is therefore considered important and necessary in many cases.

A background on the trends and previous work that has motivated this thesis' exploration of application-transparent fault management, and specifically, fault tolerance, is described here. This is followed by an overview of the goals and results obtained in this work.

1.2 Background

A number of operating systems [4], [14], [16], [33], [35] have had policies in their specification to detect, contain and recover from faults. The policies have always been integral parts of the systems, usually satisfying a narrow range of dependability and availability needs which, as an inherent assumption, all clients of the operating system desire. The current trend in operating system design, however, is not to provide a custom design for each class of computer or application environment, but rather to design one operating system that is applicable across the entire range of computer systems, from PC's to high end multiprocessors. Current examples of such operating systems include Mach 3.0 [13], [29] and Windows NT[10]. The application domains that these scalable systems address are diverse in their fault management requirements, and a one solution approach is clearly no longer acceptable. A PC used at home for personal finance management and entertainment, for example, will have drastically different needs than a computer system used as a server by a small business, though both may run the same operating system. Even the same class of computer can be used in diverse settings that have different performance and fault tolerance priorities. This indicates a need for fault management that can be dynamically tuned to the environment's or a particular application's needs.

The recent growth in popularity of distributed systems has lead to the development of software layers (middle-ware) that run on top of the native operating systems of the distributed system's nodes. Examples include Maruti [26], Consul, Isis [6] and work done at AT&T [15]. In many cases, these systems provide fault management primitives that are able to take advantage of the hardware redundancy introduced by multiple node operating environments. All of these systems provide an application with communications primitives that are used then as the basis for fault detection or fault tolerance.

Isis, for example, is based on two communications primitives, `abcast` and `cbcast`, with well defined characteristics that can be used to construct a highly available system. `Abcast` is a message broadcast that is totally ordered with respect to all other `abcast` messages. Total ordering is advantageous when calculating global predicates, determining group membership and recovering an execution. `Cbcast` is a slightly weaker, but usually sufficient, version of broadcast that preserves causal ordering. Both Consul and Maruti work on similar principals using message ordering and recording to facilitate the development of fault detection and recovery schemes.

This middle-ware approach has several potential drawbacks, however. Existing applications require varying degrees of redesign and restructuring to use the primitives and conform to specified architectural models. This fact alone makes existing software obsolete when moving to such a system. While this is of little concern in a research environment, backward compatibility is a very real, and many times decisive issue in commercial computing environments. Of secondary concern is the requirement of highly specialized programmers to develop correct and efficient programs. This is a potential additional source of introduced error. The process further burdens programmers already wrestling with hardware details, operating system quirks and the task of correctly implementing a specification with learning the intricacies of distributed programming and the middle-ware primitives. A second potential drawback is that the middle-ware usually assumes a fail-stop or fail-silent model of computation [33] which is typically not fulfilled by underlying software and hardware. A node that fails to meet this specification may invalidate the theoretical correctness of a system.

On the application-transparent side of fault management, a large number of application-transparent distributed recovery algorithms have been developed in the past decade. Several approaches have emerged including consistent checkpointing [8], [21], [24], [37], pessimistic logging [7], [37] and optimistic logging [11], [17], [18], [20], [34], [36], [37]. They have essentially evolved from the work of Koo and Toeug [21], providing a consistent checkpoint, to the work of Elnozahy [11], which provides asynchronous checkpointing and deals with important issues such as committing output. Like the middle-ware offerings, these solutions are all based on the underlying assumption that a system failure will be fail-fast or fail-silent. Computer systems usually do not have the appropriate levels of fault detection to make this assumption a valid one. This highlights the need for application-transparent fault detection policies to work with these algorithms.

An approach that has been used for both fault detection and fault tolerance is that of hardware specific solutions. A variety of computer systems, notably Tandem's S2 and NonStop, DEC's VAX 3000ft and Stratus' XA series[33], are based on hardware approaches to fault management. These systems provide extremely high levels of fault tolerance and detection, but at a high dollar price, since they require both hardware and software customizations. In Tandem's S2, the processor is triplicated and connected to its peripherals with busses that vote on results generated by each processor to detect faults. The processors are kept tightly synchronized by proprietary synchronization hardware and the fault management services are totally application-transparent. The operating system is a customized Unix that allows Posix compliant code to run on the machine without modification. Like the other operating systems used in hardware based approaches, the S2's Unix is augmented with assertions [1], [2], [16], [25] that add software error detection to the existing hardware error detection. The same overall approach is taken in the VAX 3000ft.

In the Stratus and Non-Stop cases, different hardware configurations are used. The Stratus machines use two pairs of synchronized (mirrored) processors that perform voting at two levels - between the two components of a pair, and between the two pairs. The Stratus operating system, like in the S2 case, provides transparent fault management. The NonStop takes a slightly different approach with its operating system by providing application dependent primitives to support primary/backup process pairs.

While the hardware approach to fault management can be used to provide high levels of fault detection and low-level fault masking, it does have some potential limitations. The use of customized hardware usually means that the technology these machines are based on is usually several years old. Special operating systems again mean that applications have to be tailored, at least minimally, to work on these systems. Perhaps most significantly, the cost, maintenance and complexity of these products place them out of reach of the majority of businesses and institutions that desire fault tolerance. Only industries that require the highest possible fault tolerance, such as air traffic control, airline reservations and banks, can afford them. Small businesses and individuals currently have no practical solutions to their fault management needs.

The factors discussed in this section all highlight the importance of research into the area of application-transparent policies that are software based, dynamically selectable, and efficient. The research serves to bring computing closer to the ideal goal of using low cost, high performance, general purpose computers with fault management software to provide application-transparent fault tolerance with minimal performance overhead.

1.3 Overview

An area where little research has been done is in the introduction of tunable application-transparent fault management into operating systems. Placement of fault management in the operating system is advantageous over hardware or library placement for several reasons. Since, by definition, the operating system serves as a computer's software and hardware resource manager, it has maximum visibility and control of both the application and the hardware. This combination allows for fault management policies to be application-transparent and high performance.

One common fault tolerant policy is checkpointing. If checkpointing is implemented as middleware, the application must intelligently choose what parts of its data space should be checkpointed and provide code that recovers from a checkpoint. If the checkpointing policy is placed in the operating system, checkpoints can take place without the application's knowledge and the application need not concern itself with the details of recovery or of grouping important data. Further, operating system based checkpointing can be more efficient through the use of a technique known as *incremental checkpointing* where only parts of data that have changed since the last checkpoint are saved in the new one. While this is only one example, the same characteristics apply to many other fault management policies.

The research undertaken here has a variety of goals associated with issues discussed above. The first is to define and implement an operating system mechanism that serves as the infrastructure for fault management policies. The mechanism have the following characteristics:

- be general enough to support complex fault management policies
- have an activation mechanism that provides for dynamic selection of policies
- allow an application to use multiple policies simultaneously
- be minimally intrusive

The last point means that existing operating system code must be modified as little as possible in the implementation. An operating system with a mechanism having these characteristics can be augmented with libraries of fault management policies. Since the interface to the existing operating system code is clearly defined by such a mechanism, the task of compiling versions of the operating system with different collections of policies is trivial. Different configurations can provide fault management policies matched to specific environments.

By making policies dynamically selectable and application-transparent, unmodified applications can use only the policies they require and can even select different policies depending on the environment they are being run in. A major advantage of application-transparency is that the policies need to be verified only once. Applications need not undergo fault management verification and do not have to be changed in any way to obtain the fault management guarantees provided by policies they use.

The first contribution of this thesis is in the definition and implementation of a fault management mechanism that meets these goals. The second chapter presents the mechanism, called the *sentry mechanism*, and details of its implementation.

There is not yet a good theoretical way to validate the hypothesis that a given mechanism is powerful enough for complex and practical fault management policies. The technique currently available to substantiate such a claim is through demonstration. Therefore, the definition of such a mechanism requires that policies be developed for it that are a convincing proof of the claim. Ideally, example implementations of all major fault management policies should be undertaken.

Unfortunately time does not permit this so policies have to be carefully selected. The policies chosen for development fit into a general framework investigating application-transparent fault tolerance across the spectrum of computer configurations: from single computer, to redundant computer to distributed system. The development of each policy has introduced new results in each of these areas which stand independent of the underlying sentry mechanism. The work demonstrates that the sentry mechanism is highly suitable to these environments and that the policies are complex and demanding enough that they serve as a good exercise of resources that are required by typical fault management policies.

As a prelude to the fault tolerant policies, simple monitoring and fault detection policies were implemented and are discussed with the sentry concept in Chapter 2. These policies were included in the chapter on the sentry mechanism because they serve as base-line policies and are very broad in their applicability. For example, the monitoring policy can serve roles in fault detection, system debugging, trend analysis and application profiling. Their simplicity and utility made them ideal candidates as the first policies to be implemented, but at the same time this simplicity means that they do not effectively demonstrate the sentry mechanism.

Chapter 3 presents policies that provide fault tolerance for applications running on a single computer. The fault tolerant techniques chosen for this environment were journaling, checkpointing and replay. As mentioned before, many distributed systems algorithms have been developed which address only the role of message passing in journaling and checkpointing. No known system implemented to date deals with all of the issues confronting recovery of a concurrent application running unrestricted in a typical computing environment that includes persistent storage such as a disk, and volatile input such as a keyboard and a mouse. A high priority during the development of these policies was practicality. In other words, the policies must have small performance impacts and not place any restrictions on applications running in these environments. Policies developed in this research have introduced new algorithms for single computer concurrent application journaling, as well as disk and memory checkpointing. Performance results for these policies show that fault tolerance can be provided with typically less than 5% overhead and in many cases less than 1-2% overhead.

Chapter 4 deals with multiple computer environments. Multiple computers can rely on hardware redundancy for fault masking, or on journaling and checkpointing for fault recovery. The first section of the chapter presents algorithms for dual and triple mirrored computer systems. The algorithms address the issues of internode synchronization, detection of a failed node and integration of a new node. Due to time constraints the algorithms have not been implemented, but a discussion of their integration with the sentry mechanism is presented.

The second half of Chapter 4 discusses the use of the sentry mechanism in distributed systems. During this research, the ideas developed for single computer concurrent applications have been extended to the distributed systems domain, resulting in a new distributed recovery algorithm that is superior to existing ones in certain environments. The algorithm bases recovery on the journaling of message order, rather than the approach taken by existing solutions of journaling message content. This can reduce journaling storage and performance overheads in many cases. A new form of checkpointing is developed to support order-based journaling by creating consistent checkpoints based on loose synchronization. Again, the algorithm has not been implemented, but use with the sentry mechanism is discussed.

Chapter 2

The Sentry Mechanism

2.1 Introduction

While theoretically it would be possible to place fault management mechanism interface points at every other instruction in the operating system, it is desirable to be as minimally intrusive as possible while giving fault management policies the necessary level of visibility and control to support the most complex of fault management techniques. A fault management mechanism has been developed which has the characteristics that it is minimally intrusive and provides high visibility and control to fault management policies. This mechanism has been named the *sentry mechanism* because the policies designed for it guard the operating system from undetected faults and prevent failures. This chapter presents the definition of the sentry mechanism concept, background on the Mach 3.0/UX (UNIX) system architecture [3], [13], [29], [39] and then the sentry mechanism implementation in that context. The end of the chapter covers the implementation of sample fault management policies that perform system call monitoring and fault detection through assertions.

2.2 Sentry Concept

The central idea behind the sentry concept is that operating system entry and exit points provide sufficient visibility and control to support the majority of standard fault detection and tolerance techniques. Entry and exit points of the operating system make it possible to form a *fault management layer* which sits between the application and the operating system as shown in Figure 1.

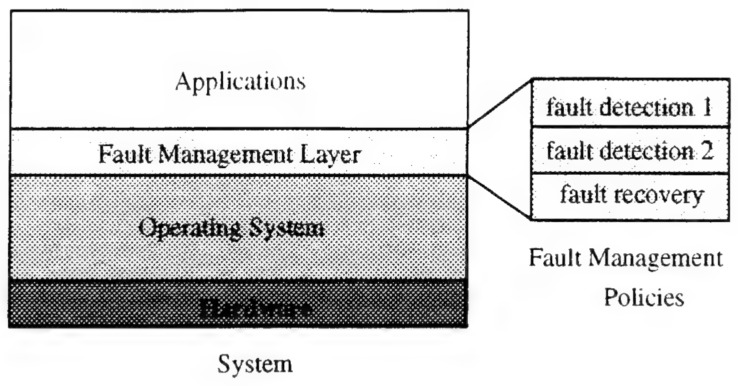


Figure 1. Fault Management Layer

These points, called *sentry points*, allow a policy to encapsulate operating system services such as system calls, page faults, interrupts, etc. This encapsulation offers sentries two powerful functions: isolation and replacement. Isolation is achieved by placing sentries that filter or change results crossing the boundary they are placed at. A bypass path from an entry sentry to its corresponding exit sentry allows for a service to be replaced by one implemented in a sentry. When sentries are enabled for a service, that service is said to be *guarded*. Figure 2 shows a service being guarded by one sentry policy.

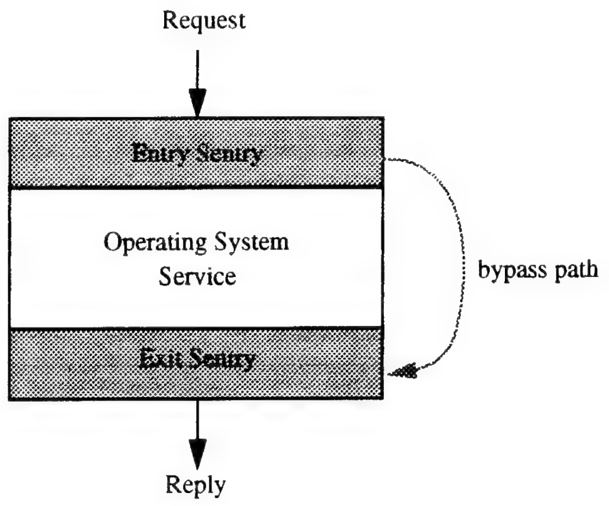


Figure 2. The sentry mechanism

One example of the importance of isolation is in fault detection/correction policies. A policy can perform consistency and validity checks on important operating system data structures, and perform correction in the case of faults, before a service is carried out that would cause the operating system to fail. The exit point allows faults to be hidden from, or reported to, an application.

System service replacement is necessary for calls that provide enhanced or modified versions of a service. For example, a fault recovery policy based on journaling will need to replicate system call behavior based on results the service provided in the initial run. The policy must provide its own version of the service that reads service results from a journal and provides them to the application, thereby recreating the initial run's outcome.

Characteristics of the mechanism include:

- the majority of fault management policies require no modification of operating system internals. This is important because modification of existing code increases the risk of introducing additional faults. Further, hard-wiring of fault management in the service precludes the advantage described below.
- policies can be dynamically assigned to guard or leave a service unguarded. This flexibility means that only processes that desire the policies will have them in place. An operating system can be implemented with sentry policies of varying scope and cost. Policies can be selected for applications at start-up or after they are running based on their specific fault management and cost/performance needs.

An operating system that has the sentry mechanism in place should provide a collection of fault management policies and an activation mechanism for selecting policies to guard specified services. Each policy should be characterized in terms of its detection or tolerance capability and the cost for each service guarded. The mechanism also allows for multiple policies to be simultaneously enabled for an application. Policies can then work in conjunction with one another. A fault recovery policy can be designed to work with a fault detection policy, for example, or complementary fault detection policies can provide increased coverage. Figure 3 depicts a service sandwiched between sentries from two policies.

2.3 Sentry Implementation

To understand the implementation of the sentry mechanism, some background describing the target operating system, which is the Mach 3.0 microkernel running a UNIX 4.3BSD server [13], is necessary. Throughout this thesis references to performance relate to the development platform which is an i486 processor (50MHz internal/25MHz external) with 16MB of main memory. Neither the sentry mechanism concept, nor any of the algorithms developed in this thesis are dependent on the Mach 3.0 architecture. This section first discusses the role of Mach 3.0 and then describes the Mach 3.0/UX architecture and the sentry mechanism implementation and activation mechanism.

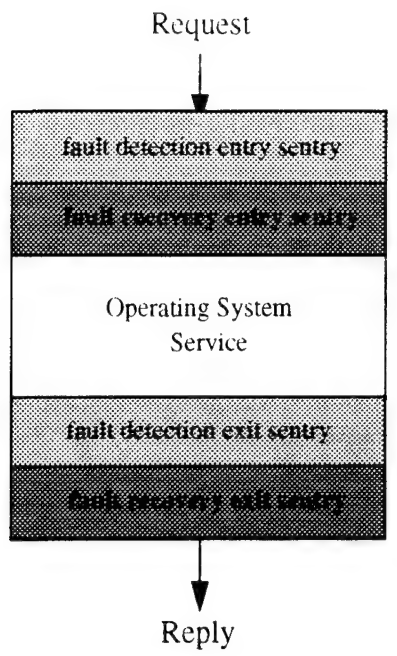

Request
fault detection entry sentry
fault recovery entry sentry
Operating System Service
fault detection exit sentry
fault recovery exit sentry
Reply

Figure 3. Multiple policies encapsulating a service

2.3.1 Does Mach 3.0 Matter?

As mentioned above, Mach 3.0 was chosen as the implementation platform. This was primarily for reasons related to accessibility, and information and debugging support. Because the system model assumed does not permit applications to make Mach 3.0 calls directly, the fact that Mach 3.0 is a microkernel based operating system did not have an affect on the sentry concept, or any algorithms developed in the thesis. In order to concentrate on form, rather than purely performance, the microkernel was not modified in any way in this implementation. This had a performance impact which is seen in external pager usage presented in Chapter 3. To improve performance, the microkernel would have to be instrumented with sentry points that allow direct manipulation of pager data structures.

Mach 3.0 represents one extreme of microkernel based operating systems where all the high level services are placed in one address space, in this case the UNIX server. Current trends in operating system design fragment this monolithic server into smaller components. In Windows NT [10], for example, there exist several servers that are dedicated to different tasks. One of these sub-servers is a file system server. If a fault management policy takes actions related to file system usage, this file system server must be augmented with sentry points. All sub-servers that are so modified must have access to a common stable storage device.

In summary, the sentry concept and the algorithms developed for fault management here are not dependent in any way on a microkernel based architecture. Further, the concepts can be mapped onto operating systems with arbitrary degrees of service dedicated fragmentation.

2.3.2 Mach 3.0 Architecture

The software architecture used in this research is the Mach 3.0 (MK) microkernel. In microkernel operating systems, system services are divided between a user-level server, which provides much of the visible operating system interface, and a supervisor-level microkernel that implements low-level resource management. The microkernel provides enough low-level support so that various servers can be implemented to run on top of it. For example, servers that have been implemented for Mach 3.0 include DOS, UNIX 4.3BSD (UX) and MacOS.

UNIX 4.3 [3], [39] was chosen as the initial platform for the sentry mechanism. Factors behind this decision include the fact that UNIX is widely used in the research community and that UNIX is at least as complex as either DOS or MacOS with respect to asynchronous signals, multitasking, memory management, and other operating system services. Therefore, results generated should be migrateable to these other operating systems. In general, both microkernel and UX services are available to UNIX applications. To limit initial sentry development, this work assumes that UNIX applications will only make UX requests. Subsequent development will also incorporate sentries in the microkernel.

In Mach 3.0 the microkernel provides an IPC mechanism, virtual memory management, low level device drivers, and task/thread scheduling. UX implements all UNIX 4.3 system calls, signals, the UFS (UNIX File System), and UNIX process management. Where necessary, UX calls on MK services such as task creation and deletion.

While there are several types of services provided by UX including signal delivery, system calls, and memory management, system calls are the most visible to an application. Figure 4 shows control flow for a system call. MK takes application system call traps and bounces control into a special UX portion of the application address space called the emulation library. This library takes the system call parameters and number, and packages an MK IPC message which is sent to UX. In UX, threads (light-weight processes) wait for incoming requests and after decoding the message, call the appropriate function. After the request has been serviced, UX packages a reply message and sends it off to the application. The emulation library unpackages the reply parameters from the message and control is returned to the instruction following the trap.

2.3.3 Sentry Mechanism Implementation

Different operating systems include different services, so the sentry mechanism must be tailored for the target system. Classes of services in UX which either are invoked by, or on behalf of a process, are:

- System calls
- Signal creation
- Signal delivery
- Process creation and deletion
- Interrupts

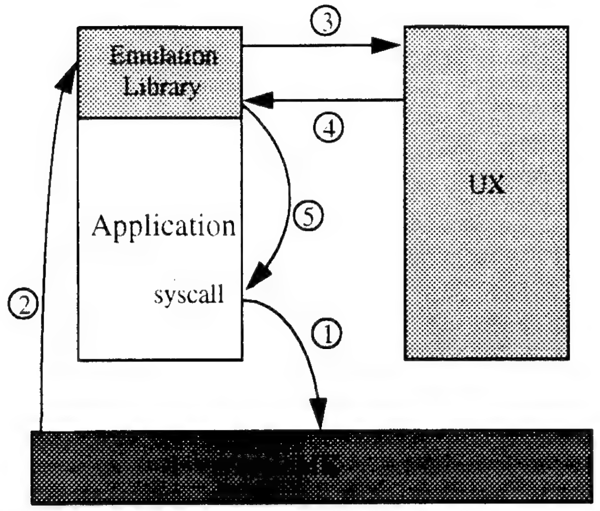
Sentry points have been placed at all of these locations in UX, and where applicable, at both the entry and exit to the service. Services are grouped into classes because all of the services of a particular class (i.e. the `read`, `write`, `open`, `close`, etc. system calls) all have the same entry and exit points. Therefore, instead of placing sentry points at the beginning and end of a `read` system call for instance, sentry points are placed in the system call handling loop.

It should be noted again here that MK offers a variety of low level services to applications running on this architecture. However, because UNIX applications that use the MK services represent a very small segment of the existing UNIX application base they have not been addressed here. When they are included in the system model, sentry points will have to also be placed at MK services.

The principle of operation for all of these points is the same, so for the purposes of demonstration, Figure 5 depicts control flow when a policy is enabled for a system call. The same path as in a standard system call is followed until the point where the UX thread receives the call request message. At that time, the thread checks to see if any policies are enabled for the requesting process for that system call service. If any are enabled, their entry sentries are executed. Entry sentries may require that the UX system call code be bypassed. When they do, control flow is passed to the code which executes enabled exit sentries (this alternate flow is not shown in the diagram).

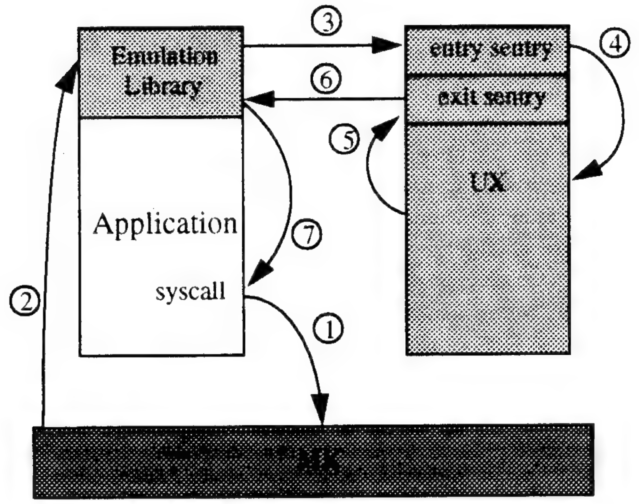
After any exit sentries are executed, a reply message is delivered to the application. The bypass mechanism, which exists at all sentry entry points, makes it possible for a policy to implement its own service in lieu of the existing operating system service code. Figure 6 shows a C code example of a entry point at a service class entrance.

At each entry and exit point there are different relevant data structures on the operating system stack that are not globally visible. This makes it necessary to create a different interface at each



1. Trap
2. Bounce back to user space
3. MK message with UX request information
4. MK message with UX reply information
5. Return to point of trap

Figure 4. Flow of control for system call in Mach 3.0/UX



1. Trap
2. Bounce back to user space
3. MK message with UX request information
4. Process system call
5. Execute exit sentry
6. MK message with UX reply information
7. Return to point of trap

Figure 5. Flow of control with sentry enabled

```
#define  NUMLIBS      XXX   /* number of sentry policies */
#define  NUMSERVS     XXX   /* number of services for sentry point */


int      sentries_on;       /* global sentry mechanism flag */
int      ftm_libsentrypt_n[NUMLIBS][NUMSERVS];
                            /* array of routines registered per
                             * policy */


/* code for sentry point 'n', servnum is the specific service
   number. For example, a read system call is assigned the
   service number '3' for the system call sentry point.
   The bypass parameter only exits for entry points and
   if set by a policy, the service is, of course, bypassed */
sentry_point_n(int servnum,int *bypass, local_vars) {
   int   i,j;
   int   (*ftm_sent)();

   /* if there are no sentries on for this process skip this */
   if(proc->p_ftmask && sentries_on) {
         /* check each policy in ascending order */
         for(i=1,j=-1; i<NUMMASK; i*=2, j++) {
               if(proc->p_ftmask & i) &&
                ftm_libsentrypt_n[j][servnum]) {
                     /* execute the sentry for policy j */
                     (int) ftm_sent =
                           ftm_librsentrypt_n[j][servnum];
                     *bypass |= ftm_sent(servnum,local_vars);
               }
         }
   }
}
```

Figure 6. Example sentry point

entry and exit point. For example, at the system call entry point, stack data that might be relevant to a policy include the system call number and the MK request message including the system call input parameters. Pointers to this stack data are passed to sentries enabled for the service, defining the interface for that point. Sentries designed for each point must meet the standard interface definition. They can then have information specific to the service being requested as well as access to all global data structures in the server.

The sentry activation mechanism includes several system calls that have been added to UX. They provide the ability to enable a policy for a particular process and for a particular service (e.g. a specific system call, signal, etc.). In the UX implementation all services are numbered. System calls are identified by their native BSD numbers and other services, such as signals, are assigned arbitrary id numbers. An example of how the interface is used will be presented after the system calls are described. Italics are specific to the complementary forms of the calls.

(*un*)`guardserv(service, policymask, policyp)` This call selects the policies flagged in the policymask to guard or stop guarding the specified service. The policyp parameter is an array that has an entry for each flagged policy. Each entry in the array contains a string and an integer. These components allow policy specific information to be passed to the policy. For example, string parameters can be used to specify a directory in which the policy should store information, or a file name. A policy specific parameter such as a mask of option bits can be stored in the integer parameter.

(*un*)`guardproc(pid, policymask)` This call selects the policies flagged in the policymask to guard or stop guarding the specified process.

`libarg(policymask, policyp)` This is a general purpose call to allow policies to be passed parameters or to execute operations dynamically. Parameters have the same definitions as those for the calls that have been described.

`sentryon(), sentryoff()` these calls are used to turn the global sentry mechanism on and off. If the sentry mechanism is off no checking is done at services for enabled sentries.

`ftexecve(command, argp, envp, policymask, policyp)` This call is identical to the UNIX 4.3 `execve` system call, but has two additional parameters for the sentry mechanism. The policymask indicates which policies should be enabled when the command is executed.

As an example of how the activation mechanism is used, assume that it is necessary to monitor all `read` system calls for an application. The user would first enable the monitoring policy for the `read` system call using the `guardserv` command. If the application is already running, the user would enable the monitoring policy for the application by executing the `guardproc` command, or, to start the process running with the monitoring policy the `ftexecve` command would be entered. When a child process is created, it inherits the enablings of the parent. The `sentryon` and `sentryoff` commands can be used to turn the monitoring off and on.

The assumptions programmed into many of UX's internal routines prevent sentry policies from using them to perform such general operations as allocating memory from a managed heap, opening and writing to files and compressing data. UX internal disk operations assume that the operation is being performed by an application process and therefore directly references the process' table of file descriptors. Sentry file operations are transparent to the application so they cannot use this table of descriptors. This necessitated the development of a library of utility routines that perform these functions independent of the context of an application process. The inclusion of this package contributes to system overhead in that additional memory operations are required to execute them. Much of the code is virtually identical to the UX code and if the UX code was made a little more general this addition would be unnecessary.

2.4 Monitoring and Assertions

To test the basic mechanism as well as develop interesting and useful policies, a system call monitoring policy and a prototype of an assertion policy were designed and implemented. To demonstrate their effectiveness, the two policies were used in conjunction with each other. The monitoring policy was used to discover the cause of a system crash that had been caused by a simple robustness evaluation program. The assertion policy was used to prevent the system crash when the program was run. This section describes the monitoring policy and its visualization, followed by the robustness program and assertion policy.

2.4.1 The Monitoring Policy

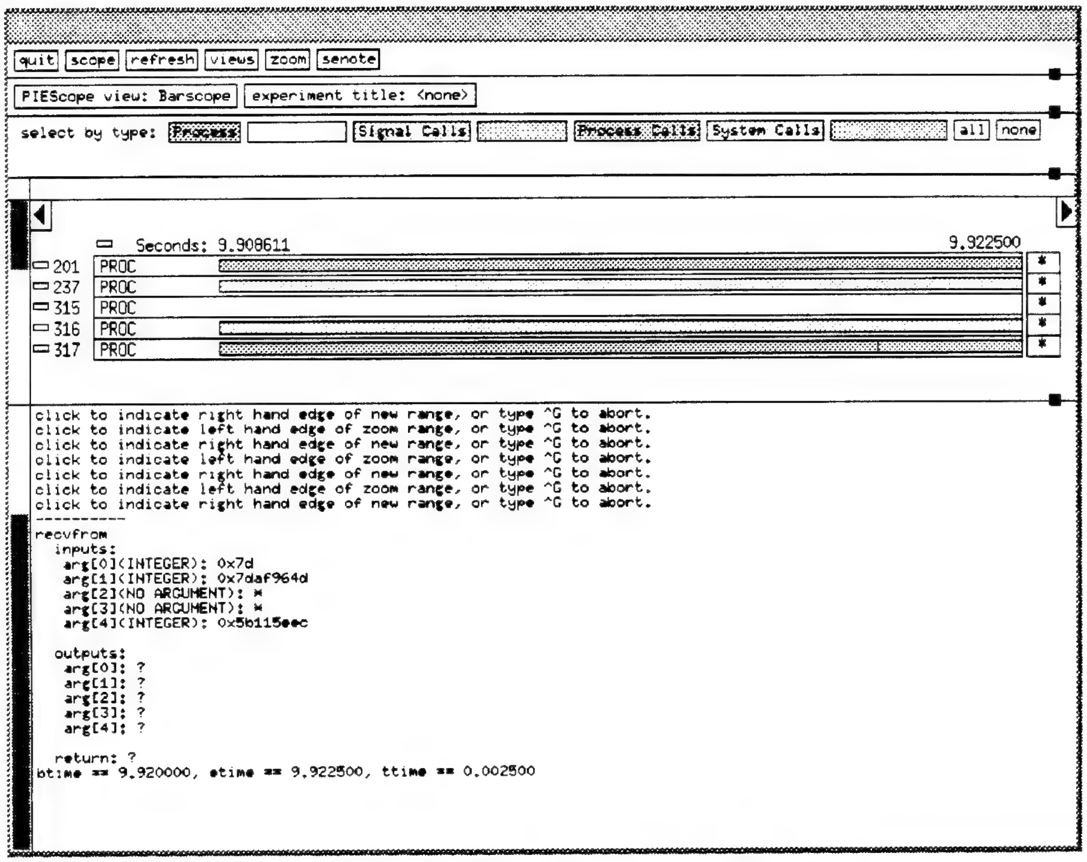
The first policy implemented for the sentry mechanism was the monitoring policy. When the monitoring policy is enabled for a system call, all of the call's input and output parameters are recorded to a file associated with the process making the call. Writes to the monitoring file can either be synchronous or asynchronous and is determined by a flag passed to the policy when it is enabled for a process. Call information can be used to debug application behavior, debug operating system behavior, analyze system performance, or can be used in trend analysis.

Several programs have been written to profile generated monitor data and calculate statistics related to performance, and frequency. As a debugging aid, a visualization tool based on the PIE [23] visualizer was designed to display the data. By using this tool, the FTSCOPE, the user is able to quickly see the system call activity of each process involved in an application and to print the input and output parameters for a selected system call.

2.4.2 The Assertion Policy

As a task related to the sentry development, robustness benchmarks are being developed to qualitatively and quantitatively evaluate system dependability. One of these benchmarks is a system call tester. This program spawns a user-specified number of processes that make random system calls, passing in buffers containing arbitrary information as parameters. This robustness benchmark program regularly crashed the implementation version of UX. To determine which system call was causing operating system crashes, the program was run with the monitoring policy enabled for synchronous writes so that information pertaining to the last system call executed would be stable on disk at the time of the crash.

The FTSCOPE view of the crash is shown in Figure 7. In the top portion of the FTSCOPE are pull down menus; below them are type buttons used to filter types of system calls being displayed as UNIX system calls can be divided into related groups. An additional type called *Error Detect* is produced by fault detection sentries when they register a fault. The portion of the window containing bars shows concurrent processes on the Y-axis and a time line on the X-axis. Each process bar is identified by its process id while different colored rectangles represent their system call activity.When a mouse is used to click on a system call rectangle, textual information is printed in the bottom portion of the window. Information includes the system call type, input and output parameters and the execution time of the call. In Figure 7 the last (rightmost) rectangle has been clicked on, revealing that it was a `recvfrom` call that never returned, showing that this was the last call made by the test program before the operating system crashed.



```
click to indicate right hand edge of new range, or type ^G to abort.
click to indicate left hand edge of zoom range, or type ^G to abort.
click to indicate right hand edge of new range, or type ^G to abort.
click to indicate left hand edge of zoom range, or type ^G to abort.
click to indicate right hand edge of new range, or type ^G to abort.
click to indicate left hand edge of zoom range, or type ^G to abort.
click to indicate right hand edge of new range, or type ^G to abort.
----------
recvfrom
  inputs:
   arg[0](INTEGER): 0x7d
   arg[1](INTEGER): 0x7daf964d
   arg[2](NO ARGUMENT): *
   arg[3](NO ARGUMENT): *
   arg[4](INTEGER): 0x5b115eec

  outputs:
   arg[0]: ?
   arg[1]: ?
   arg[2]: ?
   arg[3]: ?
   arg[4]: ?

  return: ?
btime == 9.920000, etime == 9.922500, ttime == 0.002500
```

Figure 7. FTSCOPE view of UX crash

A policy was developed that has one sentry implemented for the `recvfrom` call. This sentry performs assertion tests [1], [2], [16], [25] on each of the `recvfrom` input parameters. If an assertion fails, the call is bypassed and the exit sentry returns with an Error Detected value and a number identifying the failed assertion. Part of the actual entry assertion sentry for the `recvfrom` call is shown in Figure 8 with the first assertion test.

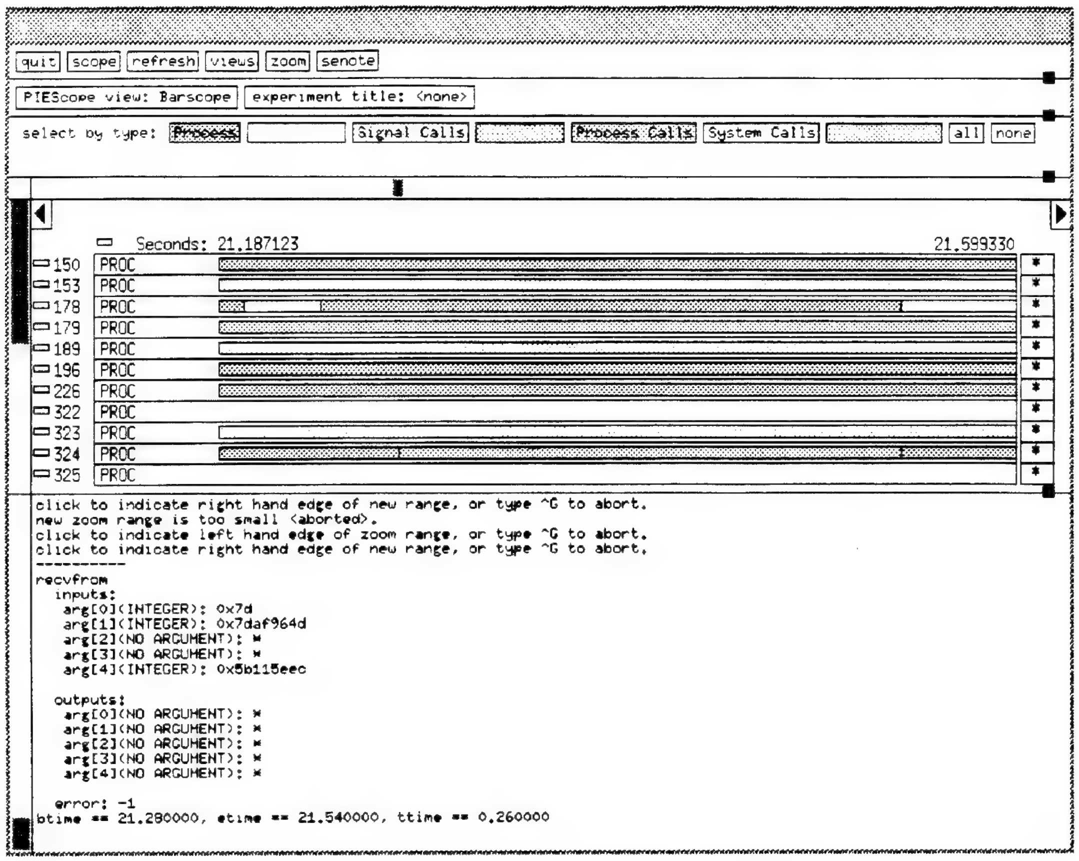
```
/*
 * parameter checking entry sentry for recvfrom
 * ftm_error is set to reflect the number of the test which
 * failed
 */
ftm_param_recvfromentry(proc_port,interrupt,req,indx,sysnum,
        ftm_error_ptr,ftm_skip_ptr)
   mach_port_t proc_port;
   boolean_t *interrupt;
   int indx,sysnum,*ftm_error_ptr,*ftm_skip_ptr;
   struct bsd_request *req;
{
   register int argoffset,fileno,flags,len;
   struct file *fp;
   int pid;
   struct proc *p;

   START_SERVER_PARALLEL(SYS_recvfrom,1)

   p = port_to_proc_lookup(proc_port);
   pid = p->p_pid;
   /* get arguments off request message */
   argoffset = sizeof(mach_msg_header_t) + sizeof(mach_msg_type_t);
   fileno = * (int *) ((char *)req + argoffset);
   argoffset += sizeof(int) + sizeof(mach_msg_type_t);
   flags = * (int *) ((char *) req + argoffset);
   argoffset += sizeof(int) + sizeof(mach_msg_type_t);
   len = * (int *) ((char *) req + argoffset);
   /*
   * first check to make sure that the socket is in range
   */
   if((unsigned) fileno >= NOFILE) {
         /* assertion 1 failed - file is out of range */
         *ftm_error_ptr = 1;
   } else {
         ...
         ...
         ...
   END_SERVER_PARALLEL
}
```

Figure 8. Recvfrom assertion sentry

When the robustness benchmark program is run with this policy enabled it does not crash the operating system. An FTSCOPE view of a run is shown in Figure 9. The call that crashed the



```
click to indicate right hand edge of new range, or type ^G to abort.
new zoom range is too small <aborted>.
click to indicate left hand edge of zoom range, or type ^G to abort.
click to indicate right hand edge of new range, or type ^G to abort.
----------
recvfrom
  inputs:
   arg[0](INTEGER): 0x7d
   arg[1](INTEGER): 0x7daf964d
   arg[2](NO ARGUMENT): *
   arg[3](NO ARGUMENT): *
   arg[4](INTEGER): 0x5b115eec

  outputs:
   arg[0](NO ARGUMENT): *
   arg[1](NO ARGUMENT): *
   arg[2](NO ARGUMENT): *
   arg[3](NO ARGUMENT): *
   arg[4](NO ARGUMENT): *

  error: -1
btime == 21.280000, etime == 21.540000, ttime == 0.260000
```

Figure 9. FTSCOPE view of avoided crash

operating system when the policy was not in effect is shown as the dark rectangle in the bottom center of the window. Its textual information shows that it has the same input parameters, and the negative return value indicates that an assertion failed in a detection policy.

This experiment demonstrates the power and efficiency of the sentry mechanism in fault detection capacities. It also highlights the usefulness of system monitoring and visualization as tools for isolating faulty behavior. The assertion policy example is somewhat trivial in that the information obtained by the monitoring library should be used to correct UX instead of writing an assertion policy, but the ideas presented are directly applicable to more sophisticated fault detection techniques.

2.5 Summary

This section has introduced the sentry mechanism as a framework of support for tunable, application-transparent fault management techniques. The sentry mechanism uses the concept of operating system service encapsulation to provide policies with high system visibility and control. This encapsulation allows policies to monitor or alter service entry or exit parameters or to replace system services through the use of a bypass facility. The chapter also presented two relatively simple, yet useful, policies, the monitoring policy and the assertion policy. Monitored data can be used for a wide variety of purposes making it a general support policy that is desirable in many environments. Fault detection through assertion testing has proven to be the method of choice in commercial fault tolerance to detect operating system faults and the prototype policy shown here demonstrates its possible application in the sentry policy domain. Finally, a powerful visualization tool was described that makes it possible to utilize monitor data to determine fault locations.

Chapter 3

Single Computer Fault Tolerance

3.1 Introduction

Since the time that computers became a personal commodity in the early 1980's, people have accumulated countless horror stories related to the loss of data and time due to the computer crashes. Papers have had to be re-typed, experiments re-run, and data re-entered. If the time and frustration that such losses have caused individuals and businesses were to be translated into dollars, the figure would surely be in the tens or hundreds of millions of dollars. Despite this fact there has been a dearth in research or commercial offerings to avoid such problems. The research here has lead to novel approaches to the problems of single computer fault tolerance, while serving as an excellent proving ground for the sentry concept.

A variety of different fault tolerance techniques including process pairs [4], [16], [33] and recovery blocks [33] have been used to provide fault tolerance in single computer environments. The most popular however, are journaling [5], [11], [12], [14], [21], [34], [35] and checkpointing [5], [12], [19], [20], [21], [34], [35], [38]. Journaling and checkpointing are complementary techniques that, unlike process pairs and recovery blocks, enable an application to survive a complete crash of the computer, including loss of all volatile data. As an application executes, information that would not automatically be recreated in a repeat run, such as inputs from the user, must be saved in stable storage. When a failure occurs, this journaled information can be used to re-execute the failed run, bringing the application up to the state that existed just before the failure. To reduce the latency of replaying a lengthy journal and to limit journal growth, checkpoints of the application's state must be periodically taken to serve as intermediate starting points. Then, when a failure occurs, the application can restart at the checkpointed state and work forward. Following a checkpoint journaled information associated with state that existed prior to the checkpoint can be discarded.

There has been no research to date that has applied the techniques of journaling and checkpointing to the software environment of modern day operating systems. Published journaling algorithms have addressed distributed systems where processes have no persistent storage and communicate via messages. This model of computation is not followed by the majority of personal computer and workstation software where disks provide for high capacity storage and communication. In addition, the algorithms are not adapted to the special environment of single computer environments where processes have tightly coupled interprocess communication.

Several high priority goals were involved in the development of the journaling and checkpointing policies for the sentry mechanism. The most important was that the policies had to be application-transparent. If this requirement were not satisfied, new programs would have be written to use the policies or existing programs would have to be modified. The majority of computer user's do not access to the source code of programs they use, and if they do, they do not have the time required to learn the intricacies of a program necessary to make modifications.

Another important goal was that the application model supported by the policies be as free of constraints as possible. It would be unacceptable, for example, to restrict programs from using disk storage, as most journaling and checkpointing algorithms do. The benchmark that has been used to measure success with respect to this goal is the ability to recover a full interactive windowed application session. This type of application encompasses all features of the modern computing environment including concurrent processes, disk storage, asynchronous signals, input and display devices.

Another high priority goal was practicality. When research began it was unclear as to what levels of overhead could be achieved in an unrestricted environment. While a target of this research was to provide policies that would make a hard disk a practical solution to stable storage, it was understood the complexities involved might make this impossible and that special purpose hardware might be required.

Finally, while recovery schemes that have attempted to address (at least in design) the scope of a modern day computing environment [7], [35] have been directed at full recovery of the machine, the approach here is to only recover a specified applications (referred to as *ft-applications*). This application oriented approach provides a system with much more flexibility as well as potentially reducing overhead. In the operating system oriented approach all processes running on a computer are involved in journaling and checkpointing and the entire operating system address space is involved in the checkpointing process. When recovery takes place a special recovery boot is necessary in lieu of the standard boot.

By focusing on a particular application, the overhead of journaling and checkpointing non-essential processes such as system daemons is avoided. Checkpointing the entire operating system is also avoided. Only data structures in the operating system that are directly relevant to the recovery of the ft-application have to be saved. This savings alone can involve several orders of magnitude in data size. Additionally, there is no requirement for a special boot sequence. After a failure the machine can be rebooted as usual, and exercised by system diagnostic programs such as UNIX's `fsck` file system consistency checker. The user can also login and perform maintenance before initiating the recovery of the ft-application. In fact, the computer can be used in normal operation with the recovery of an application post-poned indefinitely. It is important to note that the application oriented approach does allow for multiple ft-applications to be running simultaneously. In the extreme case, where all processes running on the machine are part of ft-applications, the system degenerates to the operating system oriented approach, but the advantages of normal boot and login are still maintained.

The result of this research include journaling and checkpointing policy designs and implementations that realize the above goals. Fault tolerance is provided transparently to the most demanding of typical workstation applications such as the X-window system [32] and its client programs with less than 5% overhead. The system model is flexible and allows for concurrent processes, signal delivery, disk checkpointing, file sharing, and mouse and keyboard input.

The first part of this chapter discusses the issues solved in journaling design and journaling measurements. The second part of the chapter is devoted to checkpoint design and performance.

3.2 System Model

The application model used for the checkpointing and journaling policy is a concurrent system where the application's process creation dependency graph is a tree. A node is the parent of all nodes that are below it in the tree (i.e. those nodes are its children). Processes communicate with each other primarily through message passing, but are also allowed to communicate through files and asynchronous signal delivery. UNIX BSD 4.3 does not support shared memory so shared memory issues are not addressed here. Inputs to the system from the mouse and keyboard and designated as *external inputs*. The vast majority of single processor applications fit into this model including the X-window system. An application example is shown in Figure 10.

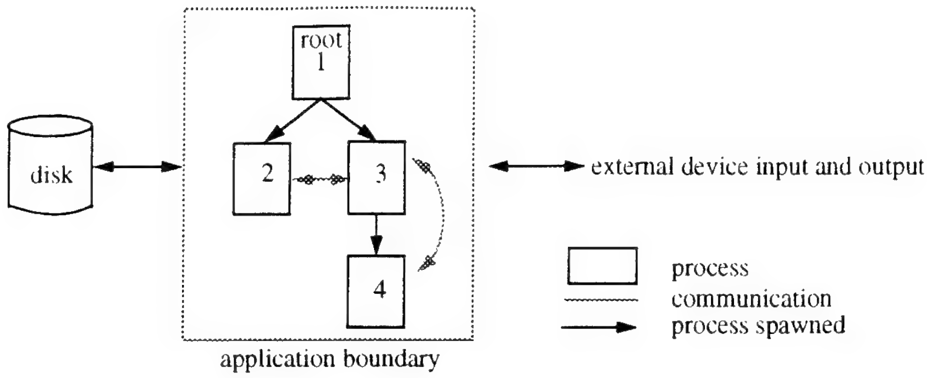


Figure 10. Example of application that fits application model

Applications that involve communication with processes that do not have the same root process do not fit this model. This means that applications are restricted to a single computer and that no communication can take place with processes on other computers or that are part of a different application tree. Relaxation of these restrictions is the focus of the chapter on multiple-computer fault tolerance.

Inter-relationships between policies and resources are shown in Figure 11.

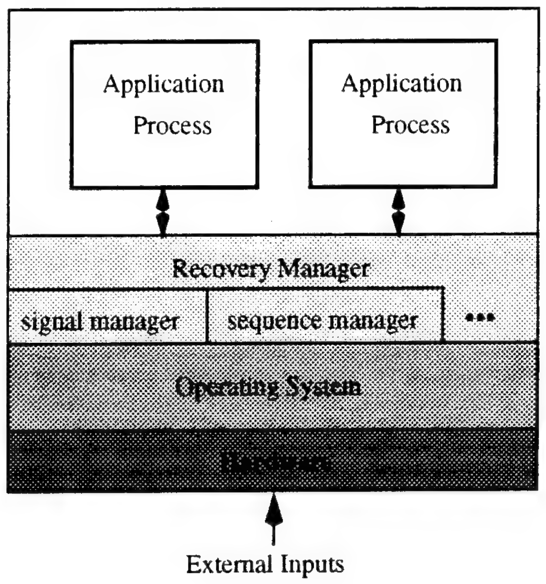


Figure 11. Architecture of single computer recovery system

3.3 Journaling Issues

Issues that had to be addressed during the development of this policy include:

- what to journal
- stable storage medium
- order of events
- journal structure
- initial checkpoint
- external input and output
- resource identifiers
- temporal related queries
- asynchronous signal delivery
- time-outs
- disk recovery
- partial recovery
- naming conventions

This section will present the approach taken for each issue in the order listed. It should be noted that there are actually two sentry policies that are related to journaling: the fault-free execution policy and the journal recovery policy. Both will be generally referred to as the "journaling policy."

3.3.1 What to Journal

One of the most important questions to be answered before the development of a journaling policy can be developed is "*what needs to be recovered?*" The answer to this question directly drives the answer to the question of what to journal. In the case of the model presented, recovery of single computer concurrent applications is desired. But what does this mean? There are several levels of recovery that can be targeted, from the two extremes of precisely replaying every machine language instruction executed up to the crash, to just being able to restart the application. The former type of recovery can only be practically achieved with extensive hardware support, and the latter does not restore any intermediate data.

Clearly, an effective solution lies somewhere between these two extremes. Therefore, data that is important in terms of recovery must be defined. In the case of the application that is being targeted, the X-window system, input from the user is considered the most important kind of volatile data.

To make this point clear, here is an illustrative example: the user has started the window system and run a complex CAD program where a detailed drawing was entered with the use of the mouse. The drawing took approximately ten minutes to input. After it was entered, the user specified that a highly computation intensive transformation be performed on the data. The transformation also takes about ten minutes to complete. Because of the length of the computation, the CAD program displays the estimated time of completion alongside the time of day clock. Sometime during the middle of the computation the computer fails due to a transient in the operating system.

If there is no fault tolerance in place the user must re-enter the drawing, a process that can be very difficult, time consuming and prone to errors. The computation could then take place without user interaction. If the fault tolerance consists of recovering every non-deterministic event that took place over the whole run, then the drawing process will be exactly replayed and the computation will display the same times displayed in the original run. While it is extremely useful to replay user inputs, it is rarely necessary for the events that followed the inputs to be replayed exactly. In this case, the original time of completion can not be valid in the later recovery run.

This does not mean that non-deterministic events can be ignored, however. For instance, time of day can play an important role when an application will make system calls prompting for input. Therefore, what is important are both all user inputs, and the record of non-deterministic events that lead up to the prompting for input. These considerations lead to a key optimization, discussed later, that makes possible the goal of practical journaling without special hardware.

3.3.2 Stable Storage

One of the requirements of journaling and checkpointing policies is that information saved about the state of an application must be able to survive a system failure. The medium used for this storage is called *stable storage,* while storage that will not survive a failure is called *volatile storage*. The choice of stable storage has an effect on the fault model (i.e. what faults the system can experience and still recover). As an example, battery-backed RAM is often used for stable storage. The RAM's contents will remain intact through a power failure, and most operating system crashes, but a failure that causes the memory to be overwritten will prevent a recovery from being possible.

Because one of the goals in the development of these policies is to provide fault tolerance without requiring the use of special hardware, the logical candidate for stable storage is a hard disk. Hard disk drives are an attractive alternative to battery backed RAM or flash memory because they are widely available and extremely cheap in terms of dollars per megabyte. Like battery backed

RAM, data on a hard disk can survive power failures as well as the majority of transient and permanent CPU failures.

Failures that are not tolerated are permanent software failures in the operating system (bugs that appear every time the application is run). This is not too dissimilar from battery backed RAM which will not tolerate transient failures that cause the data stored in it to be destroyed. Of course, installing a mirrored disk (an ideal policy for the sentry mechanism) overcomes the problem of a permanent disk failure. The strong advantage battery backed RAM has over disk storage is performance, but the algorithms introduced here make the disk a very practical stable storage device for a wide class of applications.

3.3.3 Event Order

External input alone is not sufficient to force correct recreation of a concurrent application during recovery. Preservation of event order is also necessary. In a single process environment the order of events internal to the process will always remain the same provided the process is given the same inputs. When moving to multiple processes the problem is identical to the problem addressed by distributed systems journaling algorithms. In this case the order of interprocess events must be preserved. In UNIX, events with interprocess effect are made up entirely of interprocess communication system calls (signals are handled specially, and are discussed later).

Execution of these calls creates a communication dependency graph for the application. An example execution is shown Figure 12(a). The communication pattern creates the dependency graph shown in Figure 12(b). Journaling algorithms in distributed systems are required to maintain these graphs which are used during recovery to replay events in a correct order. Parts of the graph that are not dependent on one another can be replayed by these algorithms in parallel. However, maintaining these dependency graphs becomes expensive when there is a high degree of communication.

An alternative approach is to impose a total time-precedence based order on events disregarding their dependencies. Distributed recovery schemes do not use this ordering because it requires a notion of global time, which is too costly to maintain to be practical in a distributed environment. On a uniprocessor, however, precedence ordering is easily obtained by maintaining a counter. Each interprocess event is atomically identified with a sequence number. Figure 12(c) shows this identification for the example events. These identifications are known as the *global sequence number*, and each application has a global sequence counter which is shared by all of the processes in the application. If events are replayed with the same global ordering, then all dependency graphs will be automatically recreated. A proof of this is given below.

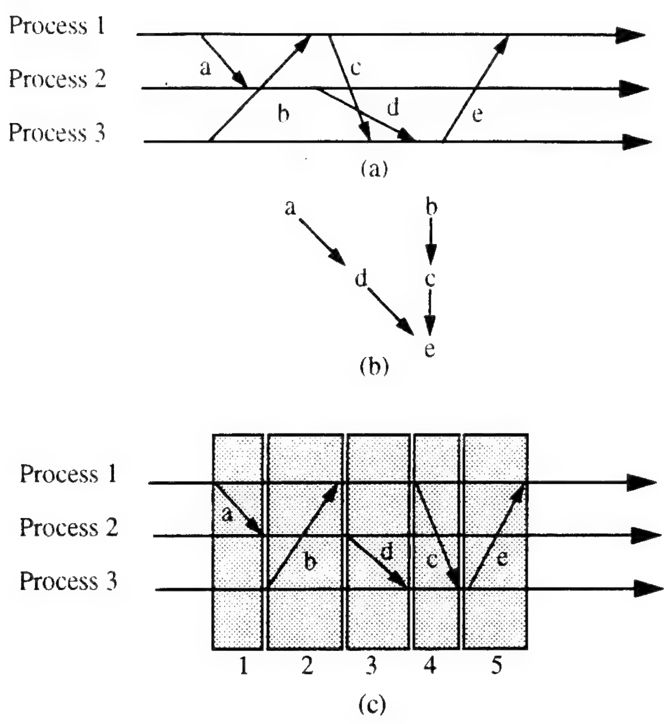


Figure 12. A) Communication pattern B) Dependency graph C) Sequencing

Theorem 1. *Recreating the global sequence will recreate all events.*

Proof. Let "$a \rightarrow b$" denote that event b "depends on" event a, and seq_e be the global sequence number assigned event e. The assumption is that if all events e such that $e \rightarrow e'$, are recreated then e' will be recreated. The proof is by induction on e. For all events $e \rightarrow e'$, global sequencing implies that $seq_{e'} > seq_e$. In replay all events e will therefore take place before e', thus recreating e'. ■

Atomicity must be enforced between assignment of a sequence number and the actual event being assigned, so that the event's sequence number is an accurate identification of its time ordering. In many cases the locking in UX does not guarantee atomicity when the sequence number is incremented in a sentry. For these situations, enough of the service must be replicated in the sentry so that locking can be added to enforce the required atomicity.

Figure 13 shows an event chart on the left for the events of an original run. On the right side it shows how event order is preserved during a recovery. In the original run, process B must wait to execute the system call event and the associated sequence increment, until A executes its atomic

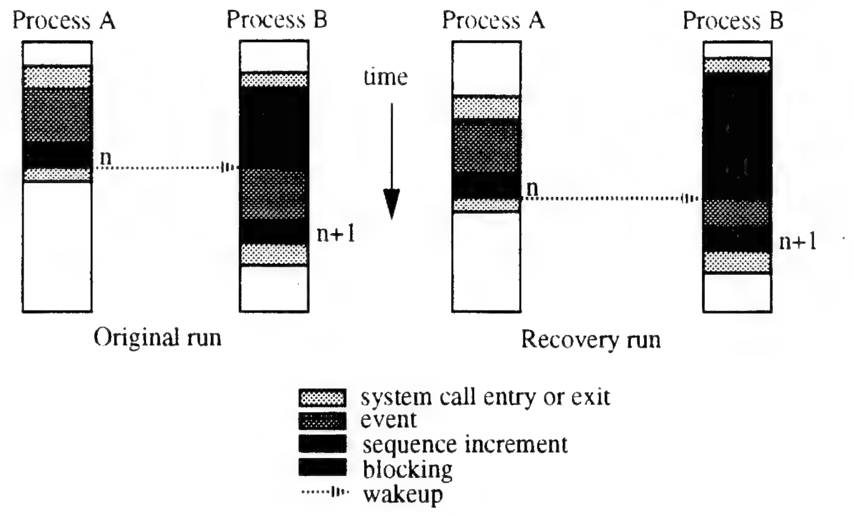


Figure 13. Event sequencing and its enforcement during recovery

event and sequence increment. During recovery, process B enters its system call first, but it must wait until the sequence number becomes *n* before it can continue.

Because the model allows for communication through files, all file related system calls must be sequenced as well. A common example of file sharing is looking at the contents of a file while it is being created by a different program.

3.3.4 Journal Structure

There are several options for structuring a journal. For example, each process of an application could have a separate journal, or the entire application could share the same journal. Each has its own advantages.

If separate journals are kept, each process can know what its next sequenced call will be by reading the next entry in its journal. In a single journal implementation synchronization must take place when data is written to, or read from, the journal. A single journal is advantageous if writing to stable storage is transactional in nature, and buffering in volatile memory or a stable storage cache can take place. A flush of independent journals means that several transactions must take place. This can lead to performance degradation associated with multiple transactions or disk operations, if a disk is the stable storage medium. For these reasons, the decision was made to have one journal shared by all processes of an application.

As mentioned above, a single journal approach introduces the problem of locating a piece of needed data. For instance, when a system call occurs during a recovery that has external inputs that must be retrieved from the journal, the sentry must find the place in the journal where the information is stored. The approach taken is to extend the sequence numbers assigned to interprocess events to all events stored in the journal. When writing to the journal a process must wait until all records with lower sequence numbers have been written before it can proceed. Then in recovery, processes executing calls that have records in the journal block until their turn to increment the sequence number. When the sequence number is incremented, the process doing the increment reads the next entry out of the journal. Entries must be tagged with process identifiers so that a process can inform the process owning the subsequent record that it is its turn to proceed. The corresponding process can then read the journal when it arrives at the point where it must do so.

Optimization of journal size can be made by noting that there is unnecessary information in the journal. Processes frequently make a series of entries in the journal that are uninterrupted by other processes. Such a series is known as a *run*. Records after the first entry of a run can be removed from the journal without loss of information, because during recovery the process that owns the first entry of a run can assume it is responsible for the missing entries as well. Note, however, that entries that have data such as device input cannot be removed.

Journal entries that are missing are initially written to the journal, but are then over-written by subsequent entries of a run. This is required to assure that an application knows when a recovery is complete and normal execution can begin. If the records were never stored in the journal, the application could erroneously conclude that recovery had finished, and then cause a different global ordering. Overwriting insured that the last record of an execution will always be in the journal.

To better demonstrate how journal run-length compression works, see Figure 14. Process A begins a run that has its first record written to the first record of the journal. Subsequent entries of its run are all written to the second record entry. When process B starts its run, its first record is written to the second entry and subsequent records are written to the third entry.

The journaling algorithm is shown in high level pseudocode in Figure 15. The primitive `wait()` causes the process to sleep until the specified condition is true while `wakeup()` allows sleeping processes that had been waiting on the specified condition to continue.

Figure 16 shows the replay algorithm. The current record is initialized at the beginning of recovery by reading the first record out of the journal.

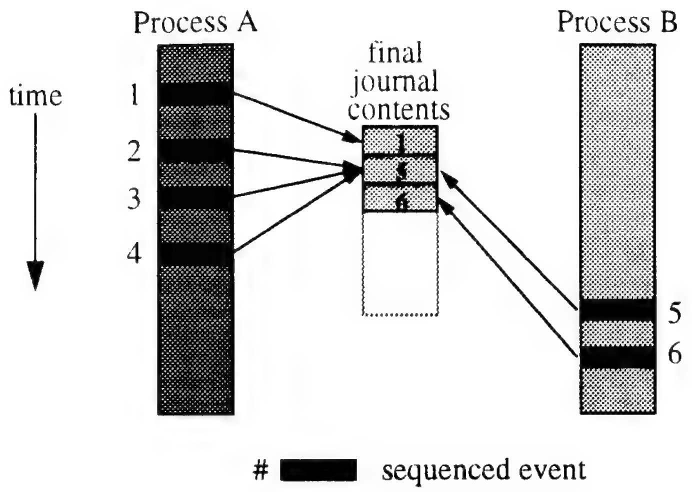


Figure 14. Run length optimization

```
routine journal(record)
   begin /* journal */
   global sequence number := global sequence number + 1;
   if(last sequence number journaled ≠
         global sequence number - 1)
         begin /* if */
         wait(last sequence number journaled =
               global sequence number - 1);
         end /* if */
   if(last entry in journal was made by this process)
         begin /* if */
         overwrite previous record with record;
         end /* if */
   else
         begin /* else */
         append record to journal
         end /* else */
   wakeup(process waiting for this sequence to be journaled);
   end /* journal */
```

Figure 15. Journaling algorithm

3.3.5 Initial Checkpoint

Journaling cannot exist without a recoverable initial state. In the absence of a checkpoint the initial state is an *implied initial checkpoint*. To recover a session the recovery policy must have some way of knowing what command began the session and must be able to present the application with the same working environment. In UNIX, for example, the recovery must start the same pro-

```
routine replay(current record)
   begin /* replay */
   if(process that made current record ≠
         this process)
         begin /* if */
         wait(current record process =
               this process);
         end /* if */
   if(global sequence number =
         current record sequence -1)
         begin /* if */
         current record := read next journal record;
         end /* if */
   global sequence number := global sequence number + 1;
   if(current record sequence number =
         global sequence number)
         begin /* if */
         current record := read next journal record;
         wakeup(process that made current record);
         end /* if */
   end /* replay */
```

Figure 16. Replay algorithm

gram, passing it the same parameters, and it must also start the application with the same working directory and environment variables. The simple solution to this is to store required information in a journal header when an application is started. The recovery process reads the header to obtain these parameters, initializes the program's environment, and starts the recovery.

3.3.6 External Input and Output

Information needed to recover an application includes any external input the application has received. In the application model described earlier, the only information received from an external source are device inputs that are not automatically and identically recreated when an application is re-run. A mouse and a keyboard are examples of these input devices, because during a recovery the same inputs cannot be made to automatically appear on these devices.

These external inputs are the relevant information to be journaled in the single computer environment. Lost mouse and keyboard inputs will not be automatically regenerated, while lost communications patterns can be recovered, albeit not precisely, without using a journal. This can be clearly seen by considering the X-window system. In the absence of journaling, if the user presses the mouse button, pulls down a menu, and makes a selection, and the computer then fails, the user's input will be lost. If the input is replayed with the aid of a journal, however, the actions that the window system takes after the menu selection will be recreated, regardless of whether it was journaled or not (note that additional inputs are ignored during recovery).

If every journal entry were important, the journal would have to be synchronously flushed every time an entry was created. Synchronous disk operations can severely degrade a system's responsiveness and throughput thus, the realization that only external inputs are important leads to the optimization of synchronously writing the journal to disk only when an external input is written to it.

Even when a system is designed to synchronously write external inputs to disk, input can be lost. If the system crashes while an input is being written to disk, an input will be lost. Input can also be lost if the rate of input exceeds the rate of disk operations. For example, a burst of typing can result in keyboard input being buffered by the keyboard device driver. If the system fails, all buffered keystrokes will be lost, regardless of the journaling strategy. Since there can be no *guarantee* made as to how many inputs can be lost in the event of a failure, the goal of the journaling policy should be to minimize the potential loss, while maximizing performance.

The fault tolerance demands of the system dictates the journaling/performance trade-offs that can be made. In some systems outputs made to certain devices must be committed [11] so that in a recovery the system will not enter a state inconsistent with outputs it has previously made. In such an environment the journal must be flushed before these external outputs are made, ensuring that the state that lead to the output will always be recreated in a recovery. In other situations, such as the one presented in the system model used here, there is no need for committed output. These systems are called here *soft fault tolerant system*s, whereas those that require commitment of certain outputs are called *hard fault tolerant systems*.

In a soft fault tolerant system, journaling/performance trade-offs are possible. If external inputs are *asynchronously* flushed, rather than synchronously flushed, the system can continue to work while the journal is being written to disk. This can improve the system's responsiveness while still maintaining a very low probability of losing more than one input. If the input rate is less than the time of an asynchronous disk operation the system is balanced as shown in Figure 17(a) . In the figure vertical arrows represent external input arrivals, and the grey rectangles represent disk operations on behalf of the numbered input. If the input arrival period becomes less than the time to perform asynchronous disk operations the system will become bottlenecked writing blocks to the disk (Figure 17(b)).

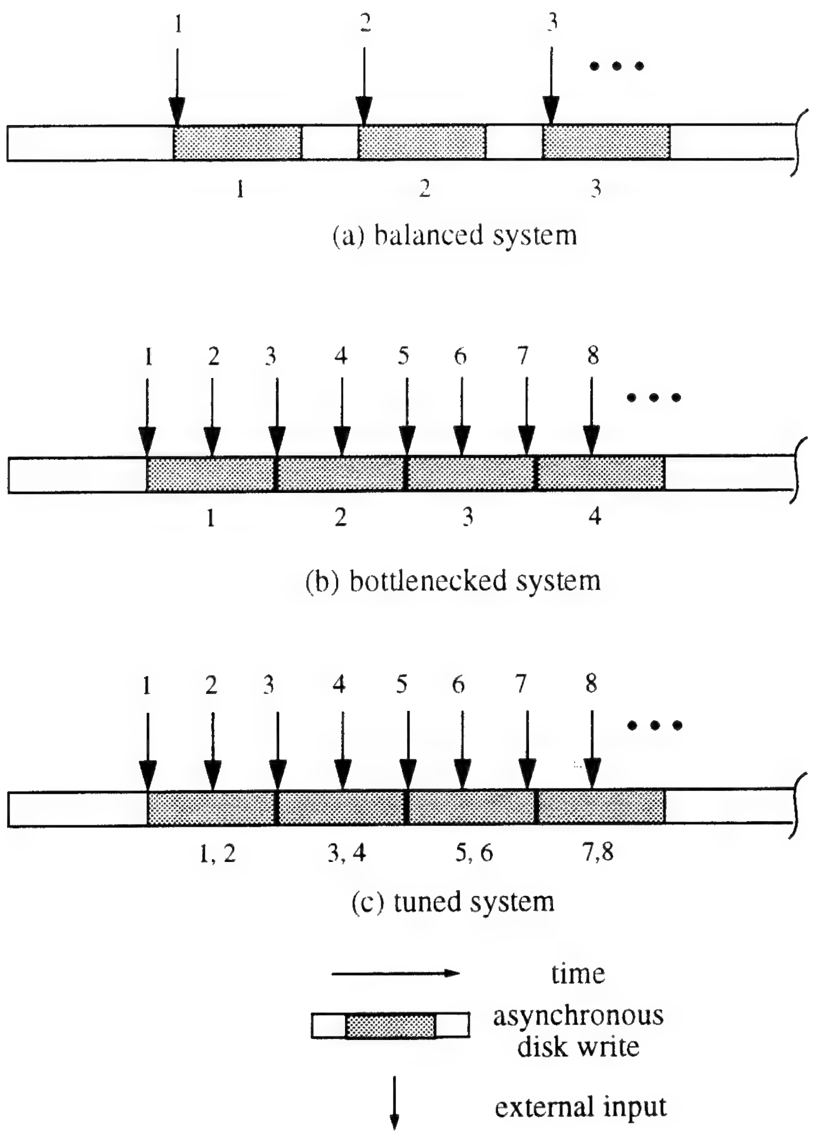


Figure 17. Different external input journaling flush strategies

It is clear that the number of inputs that are potentially lost in a failure increases as long as the bottleneck exists. To prevent this, the system can be tuned by flushing the journal every *n* inputs where *n* is determined by the following equation:

$$n = \left\lceil \frac{\text{duration of asynchronous disk operation}}{\text{input arrival period}} \right\rceil$$

In the case of a failure at most n inputs will be lost. Figure 17(c) shows a tuned system. In a typical single computer environment, the loss of a minute of input would be tolerated. This is the equivalent of several thousand mouse inputs or several hundred keystrokes. In the development system (a 50MHz i486) an asynchronous disk operation takes in the order of 50 milliseconds while the maximum arrival period of mouse inputs is about 25 milliseconds. The above equation yields $n = 2$, so to be tuned, the journal should be flushed every 2 mouse inputs. For keyboard input, the keystroke arrival period for a fast typist is about 100 milliseconds (100 words/minute where a word is 6 characters), resulting in an n of 1. The journal should therefore be flushed every keystroke. A system with these parameters will lose a maximum of two inputs that were recognized by the operating system - a result that most users would find quite acceptable.

3.3.7 Resource Identifiers

There are a variety of system calls in UNIX that act on or return resource identifiers. In the operating system oriented approach to fault tolerance, these identifiers will be identical in original and recovery runs because all data structures in the operating system, including those that the identifiers reference, will be recovered. In the application oriented approach to single computer fault tolerance, the application may not obtain the same resources during a recovery and therefore the identifiers can change.

Process identifiers are a very visible example. Process identifiers are assigned in incrementing order as processes are created. If the application recovery is started after a different number of processes have been created since the machine rebooted than in its original run, the original process identifiers might be in use (another process might own it).

Consider the following situation: a user logs in and starts a ft-application that consists of two processes with identifiers 99 and 100. After a crash, the user logs in and starts a new system daemon that receives identifier 99. The user then recovers the ft-application. The identifiers the recovered process will receive are 100 and 101 since the original identifier of 99 is in use. Forcing identifiers to their original values is therefore not possible.

A transparent process identifier mapping policy is therefore necessary for recovery. The job of this policy is to note the original identifiers and to respond to system call queries with these numbers, bypassing or changing the results of the real system query code.

Each process has two identifiers, the mapped identifier, and the real identifier. The mapped identifier corresponds to identifiers that existed in the original run of the application and may, or may not, be different from the real identifier that is returned by the operating system. The mapping policy must maintain a global notion of the mapped identifiers and must also remain active for the entire life of the application that has recovered. After a recovery is complete and a new process has been created, the policy checks the returned real identifier to see if the application perceives that identifier to already be in use (an application process has a mapped identifier identical to the new real identifier). If it is not in use, the mapped identifier is set equal to the real identifier. Otherwise to map to the real identifier, the mapping routine allocates a new mapped identifier that is unique to the mapped identifiers in use by the application. Calls that act on identifiers must pass through an inverse mapping (mapped identifier to real identifier).

3.3.8 Temporal Related Queries

Many programs have control flows that depend on system time or elapsed system time. The return values of system calls that return temporal information such as the time of day must therefore be journaled for replay during a recovery. System calls that make such queries either are bypassed with journaled values used as return parameters, or are executed and the return parameters related to time replaced with the journaled values.

If a failure occurs, a program that keeps track of time will notice a jump in time after recovery is complete, from last journaled time to actual current time, as if the program had been suspended for some duration. While this violates complete transparency of a recovery, alternatives such as providing a seamless picture of time by subtracting an offset from the actual time after recovery, have more severe problems. The two main requirements are that a recovering process sees the times it saw in an original execution and that after recovery is complete the process sees the actual time, making the current approach the most suitable.

3.3.9 Asynchronous Signals

Signals are events that consist of two types:

- self generated
- externally generated

A process self generates a signal when it performs some action that directly causes a signal to be delivered to itself. For example, executing an illegal instruction causes an illegal instruction signal delivery; accessing an illegal address causes a segmentation fault signal to be delivered, etc. Externally generated signals are delivered because of some action that another process has taken. A wakeup signal that is delivered when the process has a message arrival on a socket is one type of external signal. Another is a process kill signal that is sent from another process.

It is clear that if an application follows the same control flow, it will always create the same self generated signals at the same point in its execution. For this reason, these types of signals require no special treatment in journaling and are merely ignored.

Externally generated signals cannot be ignored. When a process sends a signal to another process during a recovery, it cannot be guaranteed that the signal will arrive at the target process at the same point in its execution as it did in an original run. In the UX implementation of UNIX external signals (other than process kill) only arrive when the target process makes a system call (note that most UNIX implementations allow for true asynchronous signal delivery where a process may be executing in its own address space when it receives a signal - a solution based on checkpointing for these implementations has been developed, but is not covered here).

If signals were included in the event sequencing discussed previously, then all system calls would require sequencing. This would result in potentially many more records being written to the journal, since typical applications execute about an order of magnitude more non-interprocess system calls than ones that are interprocess in nature and therefore sequenced. It is therefore desirable to avoid this, if possible.

The solution taken is to provide maximum control over external signal delivery by having processes execute a sentry delivering signals to itself at the appropriate times. In UX a process will recognize all externally generated signals only on a system call boundary. This means that it is always possible to replay a signal at precisely the same point in the execution of a process during a recovery. The signal delivery point is treated as an operating system entry point and thus, a sentry point exists there so that signals are visible to policies if desired.

Signals are journaled to separate signal journals that are created for each process of an application. So as not to add extra complexity and overhead to the journal flush routine executed for external input, signal files are asynchronously flushed whenever they are appended. This can mean, however, that in a recovery the journal may be inconsistent with respect to signal files. To avoid this, signals are tagged with the current global sequence number at the time of their creation. This is discussed in detail below.

When a process starts a recovery, it checks to see if a signal exists in this file and then prepares to send it on the correct system call. After a signal is delivered a new signal, if one exists, is read from the signal file. Only signals arriving on the correct system call are handled with all others being ignored. Most externally generated signals will arrive twice - once from self delivery and once from the external source - but will only be handled once and on the correct system call.

After a recovery is completed, any signals left in a process' journal with a sequence number less than the final one at the end of recovery are all sent to the process at once. Signals that have sequence numbers greater than the recovery sequence number are truncated from the signal journal. This solution works because any signals that are generated by a different process than the one receiving it are generated on a sequenced system call, either a send signal call or a communications call.

A picture of why sequence number stamping of signals is necessary is seen in Figure 18. In the original run, Process A sent a signal to Process B that was received on a system call that was not written to the journal. If after a recovery was complete, undelivered signals were simply truncated from the signal journals, the signal that is seen would be lost. By tagging the signal with sequence number *n*, the replay algorithm can see that the signal must be delivered to Process B at the end of recovery. Any signals that originated after sequence number *n+1* are then deleted from the journal.

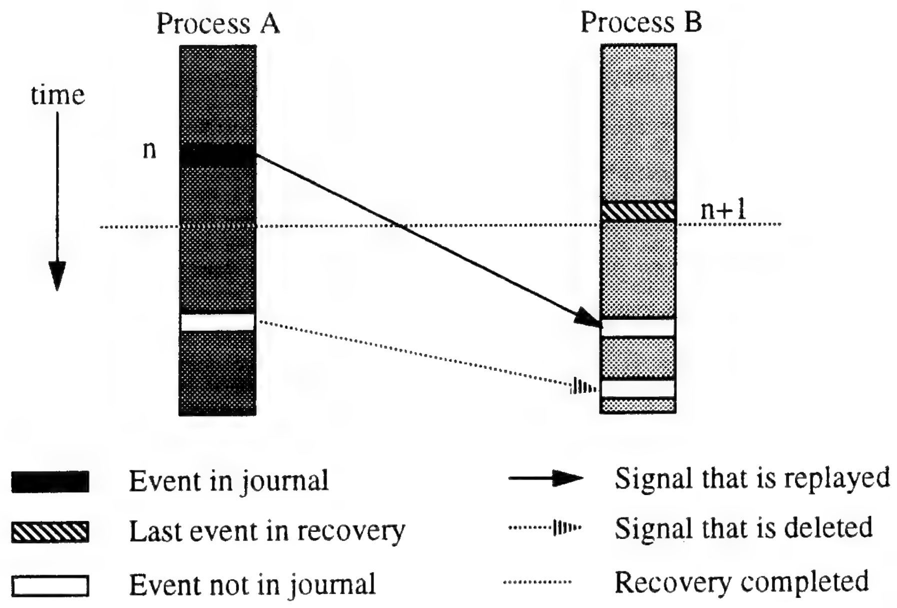


Figure 18. Signal sequence number tagging example

The "put a process to sleep" signal has to have special handling. To explain this situation consider the following: a process is put to sleep and then woken because some event occurs. In the recovery, if the process puts itself to sleep it will never wake up to deliver itself the wakeup signal. For this reason, all sleep/wakeup pairs of signals are disregarded during recovery playback.

Figure 19 presents high level pseudocode of the signal replay algorithm.

```
routine make_signal(current record)
   begin /* make signal */
   increment system call count;
   if(system call count = next signal system call count)
         begin /* if */
         send_signal(this process);
         next signal := read next signal;
         if(no more signals)
               begin /* if */
               return;
               end /* if */
         /* ignore all sleep/wakeup pairs */
         while(signal type = SLEEP)
               begin /* while */
               tentative signal := read next signal;
               if(no more signals)
                     begin /* if */
                     return;
                     end /* if */
               if(tentative signal type ≠ WAKEUP)
                     begin /* if */
                     next signal := tentative signal;
                     end /* if */
               end /* while */
         end /* if */
   end /* make signal */
```

Figure 19. Signal replay algorithm

3.3.10 Time-outs

Programs often make system calls that are of a blocking nature with a specified time-out. For example, the `xclock` program marks time by making a blocking system call with the time-out parameter set to the rate that it updates the clock display. After the time-out, `xclock` makes sequenced calls sending messages to the X server. If the blocking calls are replayed during a recovery, the time-outs will be replayed as well. Because `xclock` makes sequenced calls, the entire recovery will have to wait until the time-outs expire and `xclock` makes the calls before it can continue.

These delays can be avoided however. During the original run the journaling policy can detect time-outs and record their occurrence in the journal. During recovery, the blocked calls can be bypassed and the time-out result returned. This optimization removes unnecessary blocking from a recovery and can speed recovery orders by of magnitude.

Consider a terminal that is left idle for fifteen minutes with an `xclock` running. If the time-outs are re-executed in a recovery the user will have to sit through fifteen minutes of idle replay where the only activity is the update of the clock. With the optimization described, the idle period would be compressed to the maximum rate that the journal could be read and the clock updated. If the clock update rate is every one minute, the fifteen time-out calls would typically be replayed within seconds.

3.3.11 Disk Recovery

In the past, disk recovery has not been addressed by the majority of journaling and checkpointing algorithms. It has been investigated in the context of checkpointing, but not any known journaling algorithms. A persistent storage device such as a disk has become a cornerstone of modern computing and it cannot be ignored in any practical fault tolerant solution. When a program is using a journaling or checkpointing policy and making modifications to a disk, the modifications must be undone at the start of recovery. If the disk is not restored properly, a disk related system call might return a value different than that returned for the same call in the original run. Even one differing value can cause an application's behavior to be different and cause the recovery to fail.

Past solutions to disk checkpointing [7], [35] have been based on restoring the disk using logical disk blocks as the unit of recovery. The operating system's view of a disk is that it is a contiguous list of blocks that are allocated as needed to files and directories. Disk block checkpointing algorithms come in one of two variations. In the first variation, a logical block that is about to be modified is copied to another block which becomes a *backup* block. When another checkpoint takes place, the backup blocks are marked free and can then be allocated for another use. When a failure occurs, the disk is restored to the checkpoint by copying the backup blocks back to their originating blocks.

In the second variation, a logical block that is about to be modified becomes the checkpoint of itself. When a modification is made, the block being modified must be read into memory. The modifications are made on the in-memory copy, and if at some time the block must be written back to disk, it is written to a different block on the disk that is marked as *current*. When a checkpoint takes place the disk is brought up to date by flushing the in-memory modified blocks and copying the *current* blocks back over the original. To recover a disk checkpoint, all that must be done is to mark the *current* blocks as free.

The drawback to these algorithms is that the file system structure must be modified by introducing a new type of logical block, either the *backup* block or the *current* block. These new blocks require the addition of a reference to another block. If the current disk data structures (block header information) do not have the extra space on disk for the new reference field, then an entire redefinition of the file system layout is necessary. Also, a disk that has been modified by the algorithms is no longer readable by a standard version of the operating system or by many standard disk utility programs, and in some cases, a standard disk cannot be read by the modified kernel. This can be problematic if the disk is shared by several machines, moved from one machine to another, or the machine is booted with the standard kernel.

The disk checkpointing algorithm introduced here overcomes this problem by using files as the basic recovery unit instead of disk blocks. The algorithm is similar to the first variation above in that before a file is modified, the parts about to be modified are copied to a backup area. The algorithm requires that sufficient space is set aside for backup purposes and that a special directory be assigned for storage of backup information. It takes advantage of the fact that the only requirement in creating stable backup information is that it must be written to the disk before the actual modification is written to the disk. If the modification were written to disk synchronously, then backup information would have to be synchronously written first.

Modern computing systems use disk caches to improve average performance, so disk operations are not synchronous. In UX, blocks in the cache are allocated in a least-recently-used manner. If all cache blocks are in use and a free one is required, the least-recently modified blocks are asynchronously written to disk as the algorithm searches for an unmodified block or waits until a modified block has been flushed. This flushing policy means that data that is written first, is flushed first. Therefore, no special provision has to be made to assure that backup data is written to the disk before modifications as long as backup information is written before the modification takes place.

The algorithm must individually address each type of modification a file can undergo. This discussion is specific to the UNIX file system, but the principals are applicable to other file systems. A UNIX file, *f*, is completely and uniquely defined at any given time with the state tuple *<name,-data,stats>*. The *name* part of the file state is a textual string that uniquely identifies each file on a disk. The UNIX file system is organized as a tree with a root directory at the root of the tree. Internal nodes are directories and leaves are either files or empty directories. Components in the name separated by "/"s identify the path down the tree to the leaf node that corresponds to a file. Figure 20 shows how a name is used to locate a particular file. Components with a "/" at the end are directories. The *data* component of a file is just the actual data associated with the file, while *stats* book-keeping state such as the time when the last modification of the file took place and the size of the file.

During the course of a session a file may undergo a change in any one or all of these components resulting in the end-to-end transformation: *f:<name,data,stats>* $\Rightarrow$ *<name′,data′,stats′>*, where the right side is the state of the file at the time of a failure, and the left side is the state of the file at

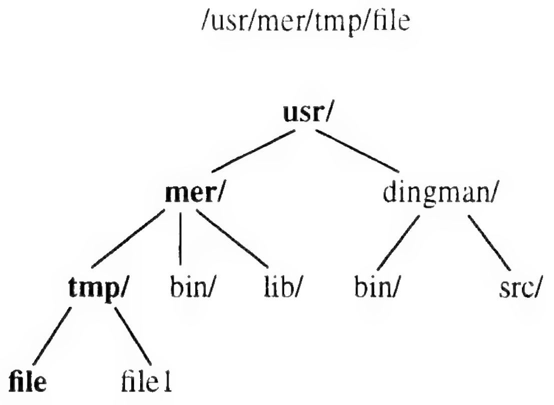


Figure 20. Example of a UNIX pathname

the time of the last checkpoint preceding the failure. To be correct, the disk checkpointing algorithm must return all files from their end states to their starting states. Each component of the tuple is restored independently and will be covered separately. Fault-free execution is covered first, followed by the recovery procedure.

Identifying each file is an important issue that must be addressed first. Since names are not stable, some other identification must be used. A simple numbering identification scheme is suitable and is made possible by the fact that disk operations are ordered by the sequencing algorithm. This means that all operations that access files will take place in the same order in a recovery as in the original run. The first access to a file that indicates that the file may be modified invokes the assignment of a monotonically increasing identifier. Since the order of first accesses to files will be in the same order during a recovery as in the original run, each file will receive the same identifier in every run. Each file assigned a new identifier has its starting *name* (this is discussed later) and *stats* appended to a *file catalog* file. It is necessary to immediately record the *stats* state in the catalog because the file access may change this.

The name of a file undergoes changes in the form *f*: *<name,data,stats>* $\Rightarrow$ *<name′,data,stats>*. A file being deleted is a special case where *name′* is *null*, and a file being created is a special case where *name* is *null*. The disk checkpointing algorithm keeps track of each file by noting the transformation in a special file called the *rename log* before the transformation is applied. While only the checkpoint name and the last name the file had before a failure are important, a log is used so that failures can be tolerated during a recovery. This is discussed in detail later.

Several special cases that must be handled. The case of deleting a file that existed at the checkpoint is handled by moving the file to the disk backup area and renaming it with its identifier number. When a new file is created its original name is set to "new," so that the restore algorithm knows the file is new and can delete it in the checkpoint recovery. Finally, if a new file is created and then later deleted, it is not moved into the backup area. The file's final name is marked as "gone" in the rename log (this will be discussed further below). A sample rename file's contents is shown in Figure 21. In the example session that created the contents, the existing file */usr/mer/tmp/file* was renamed and then deleted, and a second file, */usr/mer/tmp/file2*, was created. Finally,

```
[1] /usr/mer/tmp/file => /usr/mer/tmp/file1
[1] /usr/mer/tmp/file1 => /backup/[1]
[2] <new> => /usr/mer/file2
[3] <new> => /usr/mer/file3
[3] /usr/mer/file3 => <gone>
```

Figure 21. Example rename file

a third file, */usr/mer/file3*, is created and then deleted. The bracketed numbers on the left are the file identifiers.

The final component of a file's state that must be restored is any modified portion of its *data* state. The checkpoint algorithm keeps track of data by dividing files into fixed sized blocks (not related to file system blocks). When a file is first opened for modification, a series of bytes are allocated with one bit representing each existing block in the file. When one of these original blocks is about to be modified, the original block is copied to a *block log*. The information written to the block log is the file identifier, the starting position of the modification, the length of the data being backed and the actual data. The bit representing that block is then set so that in case it is modified more than once, it will be backed only once.

Files that are newly created do not have their *data* or *stats* states saved for recovery, but the checkpoint algorithm must record changes to their name states so that the file can be deleted in a checkpoint restore.

Recovery consists of four distinct phases. In the first phase the *file catalog* is read into memory. This sets up the file identifiers. The second phase is the rename phase. To restore the *name* components to the checkpoint, the recovery performs the inverse of each rename, starting at the last one in the *rename log*, and works back to the first. When the target of an inverse rename operation is "new" the file is deleted. When the origin of an inverse operation is the special name "gone," it indicates that this operation corresponded to a file that was created and then deleted. This means that a delete of the file must be attempted in case the record of the delete made it to disk before the delete, and then any more references to this file can be ignored. This is because the point would be to eventually delete the file and that has already been done. After the completion of the rename phase every file will be back to its starting point. Note that files that were deleted and are in the backup area are copied in the inverse operation rather than renamed. This case will be covered in further detail later.

After the rename phase comes the block restore phase. The block log file is read sequentially and the data within it is written to the file it came from. At the end of this phase every file will have its name and data states restored. This leaves the *stats* state. The original *stats* of each file were read in with the file catalog and are now applied to each file. The access times are changed to their original values and the size is also restored. The size adjustment truncates any data that was appended to a file after the checkpoint.

The only part of the restore algorithm that must make special allowance for failures is the rename phase. If a failure occurs during this phase, a file can be part way through the path it took from its final name to its original name when a failure occurs. When the recovery is restarted the rename restore algorithm will not find the file in its final position where it assumes it to be after a failure. To tolerate this case, the algorithm goes through the rename phase ignoring any errors. When the inverse operation is reached that corresponds to where the file ended up in the failed recovery, it will be eventually picked up and moved back to its starting name. Files that were deleted and were placed in the backup area must be copied instead of moved in their first inverse operation. If the files were not copied and a failure took place after the recovery, file data that had not been backed up in the block log would be lost.

During the recovery of the journal, the disk checkpointing algorithm is active exactly as it was during the original run. Not only does this dynamically recreate the disk checkpointing data structures, it is necessary for toleration of another failure. The rename file, the block backup file and the catalog file, are all preserved, but overwritten in place as the entries are dynamically recreated. The reason for this is that backup information is not synchronously written to disk. This means that when a failure occurs the data that has been flushed to the journal could be inconsistent with the data that has been written to the disk checkpoint files. An example demonstrates the potential problem: during the original run, a file is deleted. This results in the rename modification and the rename log entry being written to the disk cache. Immediately following this operation, the user enters a command on the keyboard, causing a flush of the journal, and this is followed by a system failure. If the disk checkpointing files were not rewritten during the recovery, the rename log entry would never be written to the disk. This is acceptable as long as the rename of the file does not reach the disk, but after the recovery is complete this is likely to happen. As a result, if another failure occurred, the disk would not be successfully restored to the checkpoint state because a name transform would be missing from the rename file.

During recovery system calls that query a time related stat of a file, have the stat replayed from the journal and the file is changed to reflect the time that was returned. This insures that a file's stats after recovery are the same as the application perceived them.

The disk checkpoint algorithm's fault free execution routines are shown in Figure 23 and Figure 24 with a basic support routine to allocate new file control blocks in Figure 22. The algorithms are all executed in the journal policy entry sentries for system calls that modify parts of a file's state.

```
routine new_filecb(file stats)
   begin /* new filecb */
   /* get a new file identifier */
   number of filecbs := number of filecbs + 1;
   allocate a new file control block;
   new filecb id := number of filecbs;
   hash the new filecb according to the file's name;
   if(file is new)
         begin /* if */
         record in filecb that file is new;
         end /* if */
   write filecb and id to catalog file;
   end /* new filecb */
```

Figure 22. Disk checkpoint support routine

Disk restore algorithms are shown in Figure 25 and Figure 26. The `disk_restore()` routine is called at the beginning of recovery before the processes are started.

The performance of this algorithm should be virtually identical to the first variant of the logical block algorithm because they are essentially the same. Additional in-memory data structures are required by the file based algorithm, and some extra disk space is required to manage the backup files. This overhead is small when compared to backup information and disk operations. The file based algorithm's advantage of visible backup information, and no file system modification or modification to low-level disk routines make it more versatile and practical to implement.

Proofs of the correctness of the disk checkpointing algorithm are presented below. Some of the proofs are trivial, but are included for completeness. The lemmas set out to prove that each component of the state of every file that exists at the time of a checkpoint is restored by the algorithms. Finally, a theorem combines these to prove that the entire state of the disk is restored.

```
routine file_name_modify(old_name, new_name)
   begin /* file_name_modify */
   filecb := lookup old name in hash table;
   if(filecb = NULL)
         begin /* if */
         filecb = new_filecb(old_name);
         end /* if */
   remove filecb using old_name from hash table;
   /* is it a delete operation? */
   if(new_name = NULL and file isn't marked new)
         begin /* if */
         move file to backup area;
         end /* if */
   /* deleting a file that is marked as new? */
   if(new_name = NULL and file is marked new)
         begin /* if */
         new_name = "gone";
         end /* if */
   /* is it a new file creation? */
   if(old_name = NULL)
         begin /* if */
         old_name = "new";
         end /* if */
   filecb's current name := new_name;
   append file's filecb id, old_name and
         new_name to rename file;
   hash the filecb according to current name;
   end /* file_name_modify */

routine file_data_modify(name,start,end)
   begin /* file_data_modify */
   filecb := lookup name in hash table;
   if(filecb = NULL)
         begin /* if */
         filecb = new_filecb(name);
         end /* if */
   if(filecb isn't marked new)
         begin /* if */
         if(end > file's original length)
               begin /* if */
               end := file's original length;
               end /* if */
         write any blocks between start and end that haven't
               been backed already;
         mark the newly backed blocks as backed;
         end /* if */
   end /* file_data_modify */
```

Figure 23. Disk checkpointing algorithms (a)

```
routine file_stats_modify(name)
   begin /* file_stats_modify */
   filecb := lookup name in hash table;
   if(filecb = NULL)
         begin /* if */
         filecb = new_filecb(name);
         end /* if */
   end /* file_stats_modify */
```

Figure 24. Disk checkpoint algorithms (b)

Lemma 1. *The disk checkpointing algorithm restores the name state of the disk checkpoint.*

Proof. The *name* state of the disk, D, at any given time is defined by all $f(name) \in D$. During the period between a checkpoint where the disk has the state D_c, and a failure where the disk has state D_f, the *name* state of the disk is moved through n-1 intermediate states with the individual transforms $t_i(f) = (f{:}name \Rightarrow f{:}name')$, $i = 1...n$, resulting in $D_{i-1} \Rightarrow D_i$. Given the state D_i, and the application of the inverse transform $t_i'(f) = (f{:}name' \Rightarrow f{:}name)$, the state D_{i-1} is reached. Since each transform, t_i, is recorded in the journal, the restore algorithm can work backward starting with state D_f and by applying the inverse of each recorded transform, t'_i, eventually reach state D_c. ■

Lemma 2. *The disk checkpointing algorithm restores the* stat *state of a file that existed at the checkpoint.*

Proof. Every file that has its *stat* state modified has the original *stat* state stored in the catalog file. After all other components of the file states have been restored, the files have their original *stat* state read from the file and restored. ■

Lemma 3. *The disk checkpointing algorithm restores the* data *state of a file that existed at the checkpoint.*

Proof. There are two cases to consider. The first is that a file's data has been appended with additional data. This additional data causes a change in the size of the file that is part of the *stat* state. Lemma 2 states that the *stat* state is restored, thus removing this additional data by truncating the file to its original size. The second case is that the existing data starting at position p, and of length l, is overwritten. Before the modification is made, the original data of the file $f{:}data(p,l)$

```
routine read_file_catalog()
   begin /* read file catalog */
   number_of_files := 0;
   while(not end of filecb catalog)
         begin /* while */
         read filecb from file catalog;
         newfilecb = new_filecb(file name);
         number_of_files := number_of_files + 1;
         filecb_array[number_of_files] := newfilecb;
         end /* while */
   end /* read file catalog */

routine name_restore()
   begin /* name_restore */
   move to end of rename log;
   while(not at beginning of rename log)
         begin /* while */
         read filecb id, old_name and new_name;
         filecb = filecb_array[filecb id];
         /* files marked "gone" were deleted */
         if(filecb is not marked "gone")
               begin /* if */
               if(new_name = "gone")
                     begin /* if */
                     mark filecb as gone;
                     end /* if */
               else
                     begin /* else */
                     if(old_name = "new");
                           begin /* if */
                           delete old_name;
                           end /* if */
                     else
                           begin /* else */
                           if(new_name indicates its a
                                 backup file)
                                 begin /* if */
                                 copy backup file to old_name;
                                 end /* if */
                           else
                                 begin /* else */
                                 /* standard case */
                                 rename new_name to old_name;
                                 end /* else */
                           end /* else */
                     end /* else */
               end /* if */
         end /* while */
   end /* name_restore */
```

Figure 25. Disk restore algorithms (a)

```
routine data_restore()
   begin /* data_restore */
   while(block backup is not empty)
         begin /* while */
         read fileid, start, end and data
               from block backup file;
         write to filecb_array[fileid]
               at position start the data;
         end /* while */
   end /* data restore */

routine stats_restore()
   begin /* stats_restore */
   for(index := 1 to number of files)
         begin /* for */
         restre stats of filecb_array[index];
         end /* for */
   end /* stats_restore */

routine disk_restore()
   begin /* disk_restore */
   read_file_catalog();
   name_restore();
   data_restore();
   stats_restore();
   end /* disk_restore */
```

Figure 26. Disk restore algorithms (b)

is saved to the block file. During recovery, the original data *f:data(p,l)* is read from the block file and written over modified data, thus restoring the *data* state of all files. ■

Theorem 2. *The disk checkpointing algorithm restores the state of the disk.*

Proof. The state of a disk, *D*, is defined by the state of all files *f(name,data,stats)* $\in D$. To restore the disk state from D_f, the state at the time of failure, to D_c, the state at the time of the checkpoint, every file *f(name',data',stats')* $\in D_f$ must be restored to *f(name,data,stats)* $\in D_c$. Lemma 1 states that *f(name')* $\in D_f$ is restored to *f(name)* $\in D_c$, Lemma 2 states that *f(data')* $\in D_f$ is restored to *f(data)* $\in D_c$, and Lemma 3 states that *f(stats')* $\in D_f$ is restored to *f(stats)* $\in D_c$. Thus, all three components of every file is restored, implying that D_f is restored to D_c. ■

3.3.12 Partial Replay

Journal recovery provide opportunities for tolerating "user failures" that have not been investigated previously. By only replaying a portion of a journal in recovery events can be "undone." For example, say a user inadvertently deletes important disk files or enters commands into a program that destroy work that has been done. To undo the erroneous action, the user would specify that the recovery stop immediately before the action was to be replayed. The system state would then be as if the user had never performed the action.

Specifying partial replay is more a user-interface issue than a research issue. Several different types of data can be used as hooks for replay. One approach is to artificially place timestamps in the journal and to then specify recovery as proceeding up to the end of the journal minus a time offset. Another is to use command entries as indices. The recovery could then be specified in terms of the number of commands *not* to replay. Other possibilities and variations exist with many different types of dynamic recovery control being feasible. This work does not investigate this further, but the timestamp method was chosen for implementation and served as proof of the concept.

Once a replay has finished, the journal and all disk checkpointing files are truncated at their current sizes. This erases any un-replayed events from the journal and disk checkpoint.

3.3.13 Naming Conventions

A final issue is how to organize the journal, signal files and disk files on the disk and how to specify a recovery. The naming convention is a user interface issue rather than an efficiency issue, and an infinite number of schemes could be used. To organize data for easy debugging, a special backup directory, */ftback/*, is dedicated for journaling and checkpointing. In this directory the application creates journaling and disk checkpoint directories that it tags with the root process identifier of the application. For instance, *journal.234/*, and *disk.234/* are the directories that root process 234 creates. Within the journaling directory are the journal itself, *journal*, and the signal files, like *sig.444*, which would be the signal file for process 444.

In the disk checkpointing directory are the catalog file, *fileindex*, the block backup file, *block*, and the rename file, *rename*. The recovery is started with a call to `ftexecve` that includes the recovery library sentry mask and the process identifier of the root to recover.

This naming scheme allows multiple applications to be running concurrently and to share the same backup directory without conflict.

3.4 Journaling/Recovery Examples and Performance

This section presents performance measurements taken for the journaling policy and the journal recovery policy. The overhead of the sequencing algorithms is highly dependent on dynamic characteristics of an execution. Process system call interleaving, rate of mouse and keyboard input, and amount and type of system calls are all factors influencing performance. Disk checkpointing overhead is much less variable and can be reliably measured, although the overhead caused by disk checkpointing for a given application depends on the amount, rate and type of disk operations being performed.

This section is divided into a subsection summarizing measurements of overhead for external input and communications bound applications, and a second subsection summarizing measurements of overhead for disk checkpointing. Measurements of specific actions such as repeated file read operations only serve to measure instantaneous overhead, while application session measurements serve only to give an idea of how a typical application performs. As with any technique that depends on a large number of variables such as the application characteristics, extra load on the system, hardware configuration hardware speed, and checkpoint interval, the results will vary even with the same application running on the same machine.

Before the measurements are presented, two figures are shown to provide a general idea as to the mix and types of system calls that are in a journal that has been generated by a run of X with clients. Figure 27 lists a portion of the journal from the X session experiment while Figure 28 gives a profile of records in the journal. In Figure 27, the numbers in the brackets on the left are the global sequence numbers of each record. The next number, in braces, is the process identifier of the process that executed the call. Next is the system call name corresponding to the record type, followed by any additional parameters, such as the process identifier returned by the `fork` in the first record listed.

In Figure 28 the number of records that were read from the journal is printed on the first line of the profile, followed by the greatest sequence number that was read. Statistical information below is related to the mix of system calls in the journal. The list of types of calls is sorted in decreasing order according to the total amount of space taken in the journal by the specified call type. The system call using the most space in this case is `read()` with a total of 944 records. Of these 944 records, 308 had additional information besides the standard journal entry. The additional information in this case includes mouse and keyboard input data. The right-most column lists the amount of journal dedicated to the system call in bytes (for `reads`, 13940 bytes), and the percentage of total journal size this number represents (31.2% of the journal is for `reads`). At the bottom, the total size of the journal, the amount of journal used by the standard journal record (60.92%) and the size of a standard record (8 bytes) are printed. The fact that 60.92% of journal space is for the standard records means that 39.08% of the journal is used for the storage of addi-

```
[1] <2627> fork: 2628
[2] <2627> wait: retval:8
   pid: 0 stat: 0
[4] <2628> creat
[19] <2628> gettimeofday time: 774812775.830000
[21] <2628> fstat: retval: 0
   atime: 774643031
   mtime: 760071192
   ctime: 764980760
[25] <2628> stat: retval: 0
   atime: 774797451
   mtime: 755370246
   ctime: 760065704
[26] <2628> stat: retval: 0
   atime: 774643258
   mtime: 755370246
   ctime: 760065704
[27] <2628> stat: retval: 0
   atime: 774797451
   mtime: 755370586
   ctime: 760065790
[28] <2628> stat: retval: 0
   atime: 774643258
   mtime: 755370587
   ctime: 760065790
[29] <2628> stat: retval: 0
   atime: 774797451
   mtime: 755370396
   ctime: 760065742
```

Figure 27. Example portion of a journal

tional parameters like time values and external input data. This information was useful in establishing the priorities of system call journaling optimization efforts.

3.4.1 Journaling Measurements

To date, there are no standard user-interface benchmarks so several scenarios were selected that together provide an overall view of journaling performance. The vast majority of UNIX workstations in use run the X-window system. For this reason, all measurements were made in the X environment. The application model encompasses X and any programs run under X such as `xterm`, `csh`, and `gnu-emacs`.

In evaluating the journaling policy, the cost of two components must be determined. First, interprocess communication causes journaling to record event order. Second, external device input

```
end of file - 3402 calls
done - 3402 total calls (maxglob: 9127)

------------------------STATISTICS-------------------------
   call                    number             memory usage
-----------------------------------------------------------
   read:                   944/ 308           13940 ( 31.2%)
   gettimeofday:           662/ 662           10608 ( 23.7%)
   select:                 1179/ 289          10262 ( 23%)
   stat:                   211/ 184           3896 ( 8.72%)
   fstat:                  156/ 156           3120 ( 6.98%)
   write:                  83/ 0              664 ( 1.49%)
   wait:                   20/ 20             608 ( 1.36%)
   ioctl:                  44/ 0              352 ( 0.788%)
   exit:                   22/ 0              176 ( 0.394%)
   fork:                   17/ 17             170 ( 0.381%)
   setitimer:              8/ 5               144 ( 0.322%)
   open:                   13/ 0              104 ( 0.233%)
   ftgettime:              12/ 0              96 ( 0.215%)
   execve:                 6/ 0               48 ( 0.107%)
   fcntl:                  6/ 0               48 ( 0.107%)
   close:                  5/ 0               40 ( 0.0895%)
   connect:                5/ 0               40 ( 0.0895%)
   creat:                  3/ 0               24 ( 0.0537%)
   interrupted:            3/ 0               24 ( 0.0537%)
   dup2:                   2/ 0               16 ( 0.0358%)
   globinc:                1/ 0               8 ( 0.0179%)
--------------------------TOTALS---------------------------
   total: 3402             mem: 44677         header:1155
                     base mem: 27216 ( 60.92%)
                           record size: 8
```

Figure 28. Profile of X session

causes journaling with the two types of external input on most workstations being a keyboard and a mouse. With this information in mind four tests were selected:

- X initialization
- mouse movement
- keyboard input
- X session

This section describes and presents the results runs of each test with:

1. No policy enabled
2. Basic journaling
3. Run-length optimization

In each case, times are shown for the journaling policy and the recovery policy. In addition, the size of the journal, the number of journaled calls, and the number of processes involved in the test are listed. All table time values are in seconds and all size values are in bytes representing averages of 10 executions of each test. In all cases the deviation from the average was less than 1%.

For each individual journaling test, times are shown for both synchronous journaling to the disk and asynchronous journaling. Synchronous journaling is used if full recovery of every journaled event is desired and there is no stable memory. Asynchronous journaling approximates results that would be obtained if the journal were buffered in stable memory before being written to disk.

The start-up of X involves on the order of 20 processes. X start-up occurs when the `xinit` program is run and is considered complete when an `xclock` and an `xterm` window have been created and a cursor appears in the `xterm` indicating that input can commence. Figure 29 is a picture

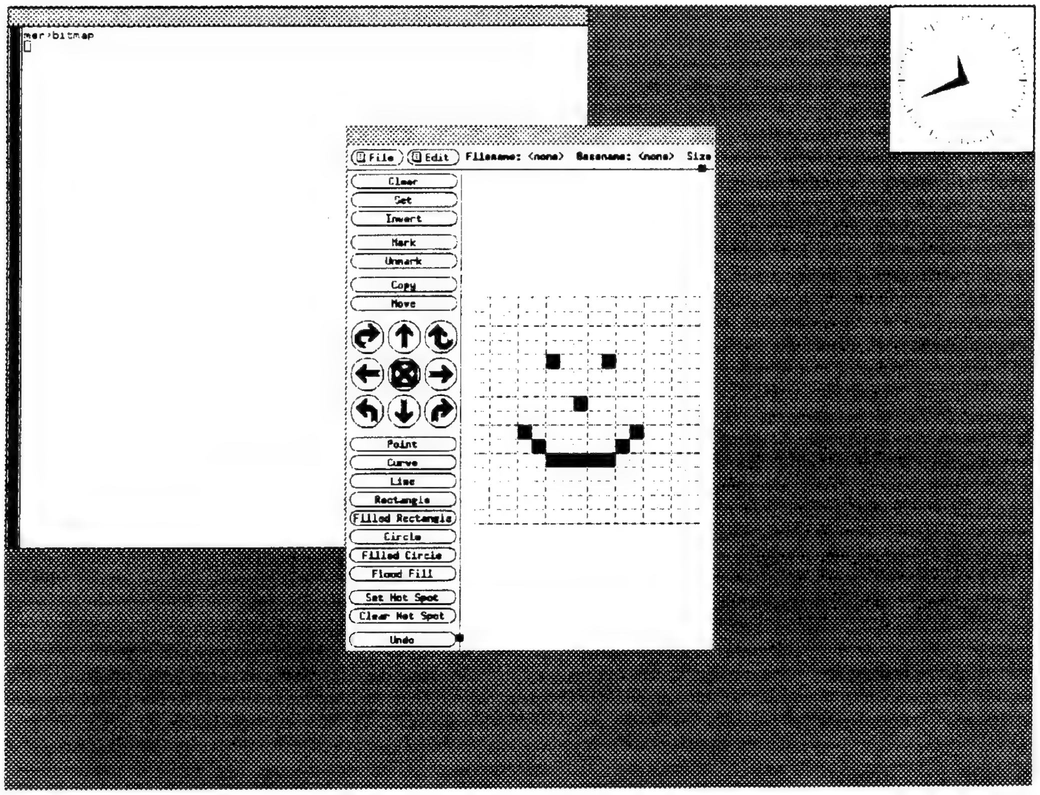


Figure 29. X startup with Bitmap application

of X after start-up where the user has run the `bitmap` interactive bitmap editor. Measurements taken in this test to showing degradation due to sequencing and event ordering of communication calls are summarized shown in Table 1. Both times and journal sizes are averaged over several runs and are rounded to the nearest second and 100 bytes respectively.

X Initialization	# of procs	asynchronous		synchronous		recovery time (sec)
		time (sec)	journal (bytes)	time (sec)	journal (bytes)	
no journaling	20	5.5	-	-	-	-
basic journaling		6.0	28100	218	28900	5
run-length opt			8900	180	22400	

Table 1. X start-up performance

Journal sizes are larger for synchronous journaling because flushing on each record causes more context switches, and therefore fewer chances for run-length compression. This effect is seen in all of the tests.

X start-up time increased from 5.5 seconds without journaling to 6.0 seconds when asynchronous journaling was enabled introducing roughly a 9 percent degradation. The measurements indicate that synchronous journaling is totally impractical since it would result in degradation of almost two orders of magnitude. The table also shows that run-length compression results in a 60% reduction in journal size.

Test results summarized in Table 2 are the result of continuously moving the mouse from the top to the bottom of the monitor screen and back for 60 seconds. Degradation effects due to journaling overhead is not so much evident in latency of moving the mouse from one position on the screen to another, but more in the number of times the mouse is sampled per unit of time. A lower sampling rate results in jerky mouse motion. Another type of journaling is introduced here called *semi-synchronous* journaling. Semi-synchronous journaling is the name given to the input related

Mouse Input	# of procs	asynchronous		synchronous		semi-synchronous		recovery time (sec)
		event	journal (bytes)	event	journal (bytes)	event	journal (bytes)	
no journaling	3	2420	-	-	-	-	-	-
basic journaling		2410	235200	32	60300	2380	233700	7
run-length opt		2420	168600	32	34300	2380	161900	

Table 2. Mouse input performance

flushing strategies discussed earlier in this chapter. No measurements are shown for semi-synchronous journaling in Table 1 because there is no input during this test. If the test were run with semi-synchronous journaling results would be identical to the asynchronous case. It is exciting to note from Table 2 that there is only about a 3% degradation for semi-synchronous journaling with respect to no journaling. Recall that with semi-synchronous journaling the fault tolerant guarantee is that at most two mouse samples will be lost in a failure. This level of overhead is virtually undetectable by users.

The last number in Table 2 is the replay time for the asynchronous and semi-synchronous cases. Replay compression results in almost a 10 to 1 reduction in time versus the original run 60 second.

The keyboard input test measures the cost of journaling and replaying keystrokes. Test data is presented in Table 3. Keyboard journaling was visible during a run as the delay between keystrokes and the appearance of characters on the screen. For each run in this test, keyboard characters were entered at the maximum rate for 60 second where the typing rate was reduced to the rate of character output. Thus, the higher the journaling overhead, the longer the time to type the characters.

Keyboard Input	# of procs	synchronous		semi-synchronous		asynchronous		rcvry time (sec)
		events	journal (bytes)	events	journal (bytes)	event	journal (bytes)	
no journaling	3	665	-	-	-	-	-	-
basic journaling		32	11300	665	164000	665	164900	5
run-length opt		40	11900	665	78600	665	79000	

Table 3. Keyboard input performance

Semi-synchronous overhead for keyboard input is not measurable and, like the mouse experiment, there is about a factor of 10 compression for the replay compared with the 60 second original run.

This final test consisted of an interactive session using elements from all of the above tests. The X-window system was started, a `gnu-emacs` editor was opened and a file edited that required a number of modifications. The editor was then closed and X exited. Keyboard input was again limited to the display rate, as in the keyboard test. This example illustrates performance of the system under practical working conditions

Measurements are shown in Table 4. In the semi-synchronous case, which provides fault tolerant guarantees, the degradation is essentially undetectable. Recovery is 6 times faster than the original run. The test indicates that on average, journal growth rates of 50-100 KB/s can be expected under typical working conditions.

X Session	# of procs	synchronous		semi-synchronous		asynchronous		recovery time (sec)
		time (sec)	journal (bytes)	time (sec)	journal (bytes)	time (sec)	journal (bytes)	
no journaling	21	60	-	-	-	-	-	-
basic journaling		424	78600	60	109700	60	100600	10
run-length opt		422	68400	60	56500	60	42900	

Table 4. X session performance

3.4.2 Disk Checkpointing

The disk checkpointing algorithm can cause overhead in the following disk operations:

- renaming a file
- creating a new file
- deleting a file
- modifying a a file

Whether or not the operation results in overhead depends on a variety of factors. For example, opening a file for modification only causes the overhead of a write to the name file the first time this operation is performed for a particular path name after a checkpoint. Also, if a file is modified, the conditions for the disk checkpointing algorithm to cause an overhead are that the part of the file being modified existed at the time of the checkpoint and that this is the first time that part has been modified since the checkpoint. For these reasons, the typical file system benchmark which creates a file, writes data to it, and reads it back, will, under most circumstances, not incur any disk checkpointing overhead.

Under all but the most extreme circumstances it can be safely stated that disk checkpointing overhead will be negligible. This is due to several reasons. A crucial one is that most applications do not modify existing portions of files. For instance, when a user edits a file with the `gnu-emacs` editor, the editor reads the existing file, performs modifications on an in-memory copy, and when the modifications are saved, it writes out a completely new file, renaming the old file to a backup name. Also, disk intensive applications are rare and usually involve appending data to a new file which does not result in overhead. Finally, modifications that do cause overhead are likely to be undetectable because they are infrequent (creating a new file, for example) with respect to the rest of the computation and due to the asynchronous nature of disk operations will occur in parallel with other operations.

While in most cases it is true that disk checkpointing will not result in overall overhead, it is necessary to obtain some measure of degree of its instantaneous overhead. A suite of artificial programs were written to stress the components of disk checkpointing mentioned above. For instance, to measure the overhead of creating a new file, a test program created a varying number of files, and the overhead per file created was calculated with disk checkpointing both enabled and disabled. Resulting curves provide insights into disk checkpointing overhead trends.

A total of five test programs were written to measure the overhead of:

- file creation
- file deletion
- file renaming
- modifying existing portions of a file

Figure 30 presents file creation test results.As the number of files created increases checkpoint time per file goes up. This is because in UNIX directory searches are linear, so that as the number of files goes up, linear search time goes up. In addition, additional disk blocks are allocated for the directory as more files are created. A typical directory will contain on the order of a hundred files or less, so the points to the left of the graph are most representative of the typical create operation. Disk checkpoint create takes roughly four times what the native file create takes.

The next tests measured the overhead of delete operations. Results are summarized in Figure 31. In these tests files that were deleted existed before the tests were run. Again, the more files that exist, the higher the overhead due to the linear directory searches. Delete operations in native mode are inexpensive because deallocation of disk blocks merely involves changing their status. This is unlike the create operation where a free block must be found and a variety of pointers allocated. When the disk checkpointing policy is active however, the absolute time is roughly equivalent to the measurements taken in the file creation operation because disk checkpointing book-keeping operations are identical for both create and delete operations.

Rename overhead is again relatively high with a rename operation for a typical directory taking about 5 times longer than a native rename as shown in Figure 32. The disk checkpointing algorithm again must do the same book-keeping as for file deletion and creation. Once again, linear directory searches cause both native and disk checkpointing times to go up as the number of files goes up.

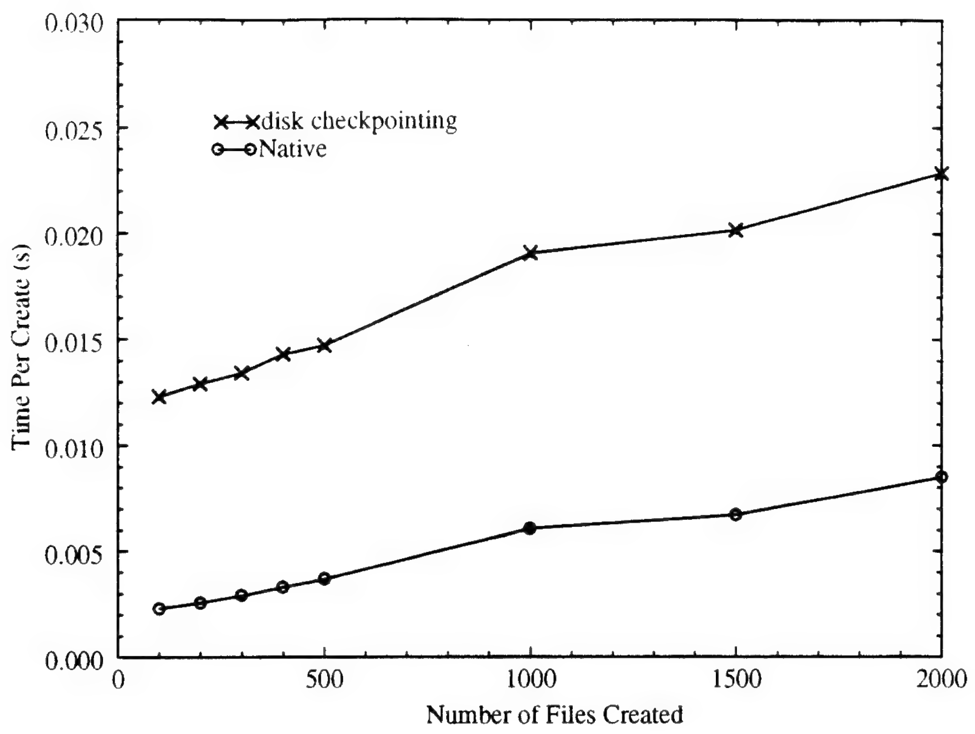


Figure 30. File creation overhead

Final tests involved modifying existing files. The test program took a 3 megabyte file and sequentially overwrote it using selected size blocks. Different runs were made with varying block sizes. The initial file was large enough so that the modifications could not be cached and resulted in actual writes to the disk. Results are shown in Figure 33. Times shown are the time required per write operation and naturally increases as the block size of the write operation increases. It increases more linearly for disk checkpointing because of the additional disk operations. For most block sizes this represents a factor of two increase relative to the times required for native file modifications.

As a closing note on the disk checkpointing measurements, it should be emphasized that the high overheads obtained (2-5) for disk checkpointing with respect to native operation times will probably be negligible in typical application sessions.

3.5 Checkpointing Issues

There are two problems that prevent journaling from being a truly practical stand-alone policy: journal growth and recovery latency. As events occur during an application session, the journal will continue to grow, and given enough events, would eventually consume all available disk

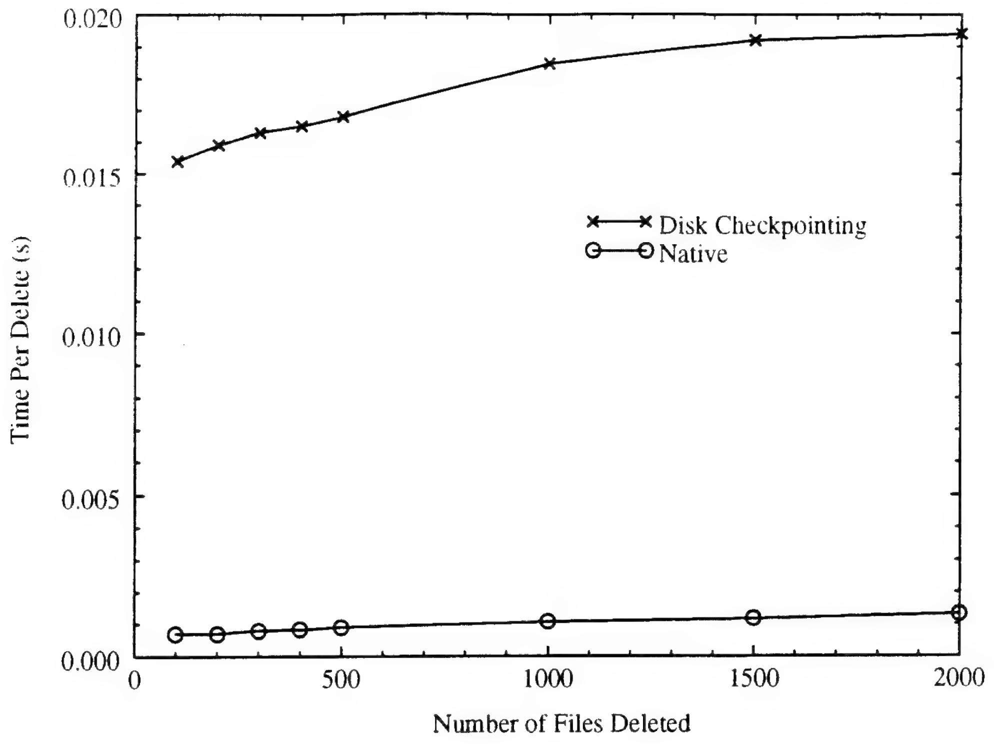


Figure 31. File deletion overhead

space. The second issue becomes a problem for long running applications. For a recovery there is no time compression for compute bound segments of code. As the length of the original run grows, so does recovery time; and, as recovery time grows, so does the probability that it will not be useful to recover the state of the application using the journal. For instance, if a compute intensive application has run for 24 hours, a recovery that takes 22 hours is likely to be unacceptable.

An appropriate checkpointing policy solves both of these problems. If an application's state is periodically saved, the journal can be cleaned. This reduces data storage requirements to the size of two checkpoints and a journal, instead of being unbounded. The amount of space allotted for the journal can be capped, such that when the cap is reached a checkpoint is automatically taken that deletes the old journal and begins a new one.

At the time of recovery the application is first restored to its most recent checkpoint followed by journal recovery to bring the application back to the state that existed at the time of the failure. If checkpoints are taken every thirty minutes, the average recovery time will be fifteen minutes or less, regardless of the total duration of the session.

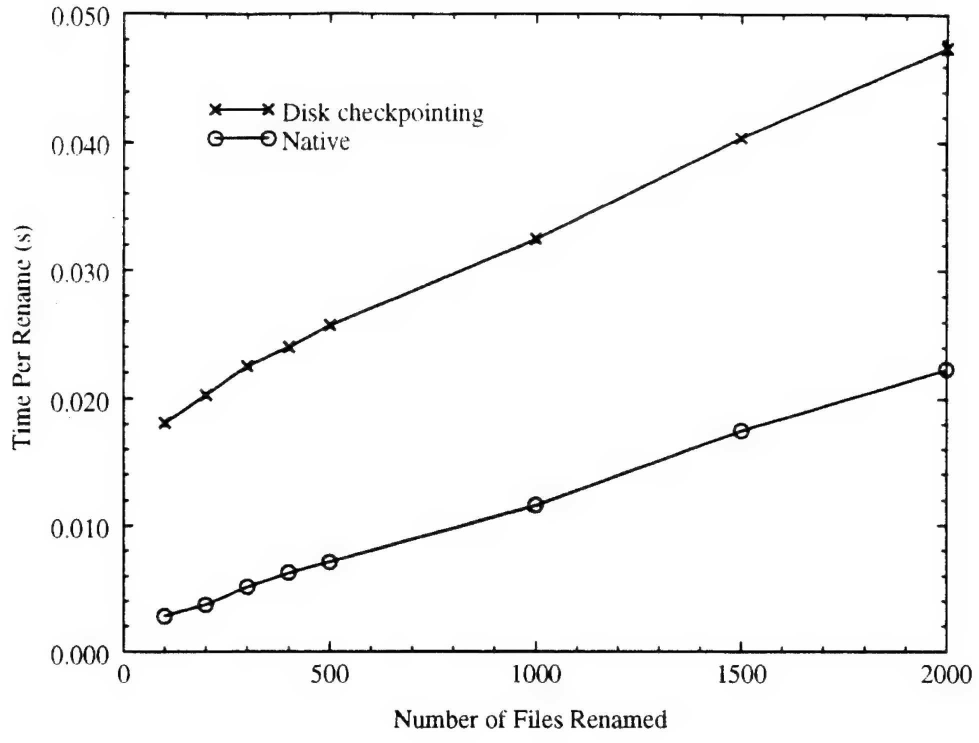


Figure 32. File rename overhead

The same application model assumed for journaling is assumed for checkpointing. As with journaling, there are several issues that must be addressed for a checkpointing implementation. They include:

- naming strategy
- memory checkpointing
- disk checkpointing
- operating system checkpointing
- taking a snapshot
- device checkpointing
- checkpoint interval
- checkpoint recovery

This section discusses each of these areas.

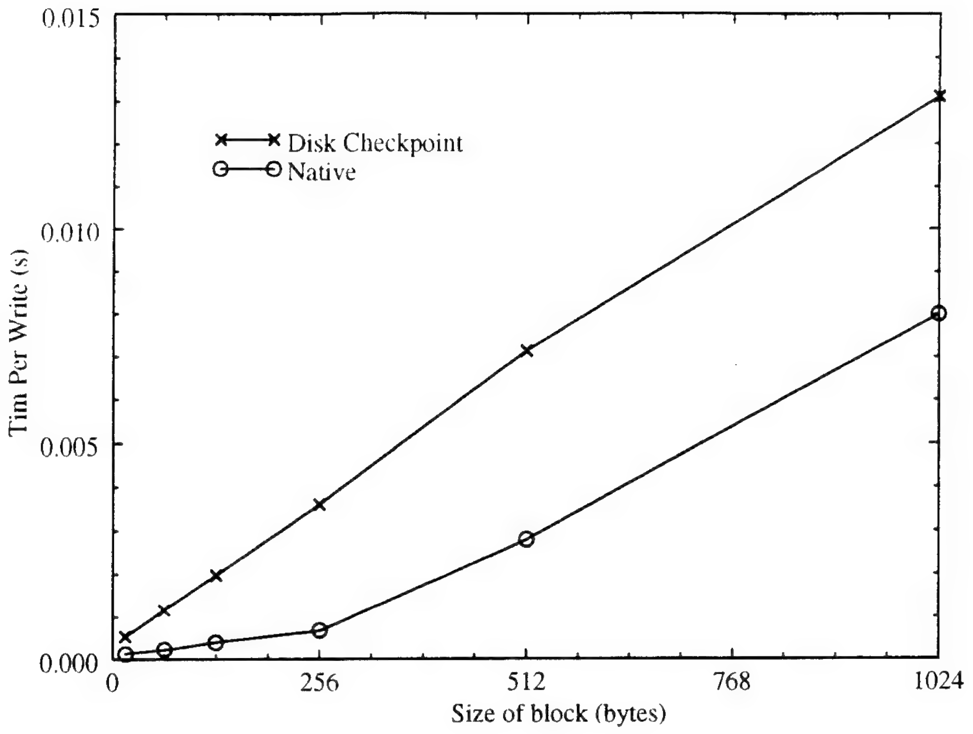


Figure 33. File overwrite overhead

3.5.1 Naming Strategy

This is the easiest issue to deal with. Naming refers to the convention used to identify files that are related to different checkpoints, devices, and processes. The approach taken is to organize all process related checkpoint files into one directory that is tagged with the application's root process identifier. For example: */ftback/mem.432/*. Device related files are also placed in a separate directory like */ftback/dev.432/*.

The strategy taken for individual checkpoint files is to add the process identifier the file is associated with to the name of the file, and add as a suffix the checkpoint interval. For example, the second checkpoint of the process control block of process 768 would have the name *proc.768.2*.

3.5.2 Memory Checkpointing

Modern applications have become very memory intensive compared with the applications of just a few years ago. Applications such as X, that are made up of many processes, can have on the order of several megabytes of virtual memory in use at a given point in time. Saving the entire address space of each process at every checkpoint would be extremely inefficient. Fortunately incremental checkpointing [35], a technique where only those parts of the system that have *changed* since the previous checkpoint are saved, can be used instead, substantially reducing disk space and performance impacts.

Mach 3.0 allows a process to use a pager that is external to the kernel to manage its memory. The checkpointing implementation relies on this feature by adding memory management to UX. Processes run with the checkpoint policy in effect use the UX pager to keep track of page modifications.

An implied checkpoint exists at the start of an application. It includes disk checkpoint and the program command, parameters, and environment variables that started the application. This is the same as the initial journal checkpoint. After a process has started, the pager keeps track of any modifications on a page by page basis. At a checkpoint these modified pages are saved to stable storage. The checkpoint is complete, or *committed*, when all modifications have been saved. At that point the journal can be cleared and the previous checkpoint erased.

At the time of recovery, a process is loaded as it would be if it were started for the first time. Its address space is then allocated and copied from the checkpointed memory information. After this has been done for each process in the application, the entire memory space of the application has been restored.

The stable storage medium plays a role in determining how to organize checkpoint data. In a disk based system such as the one implemented, practicalities have to be faced. Ideally, a separate file would be dedicated to each page. Unfortunately, this is not feasible in the UNIX file system.

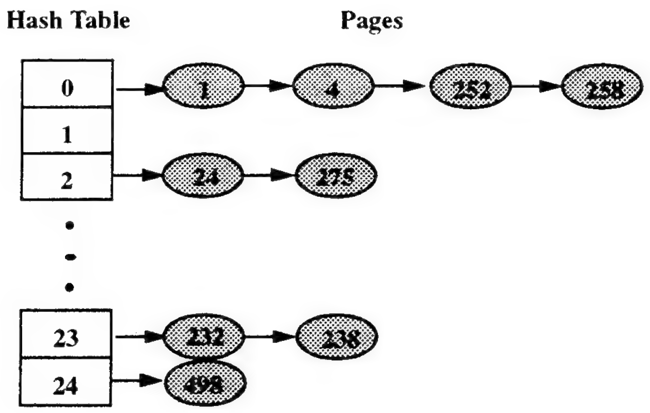
Consider an application that uses 6 megabytes of memory. In the typical system the page size is 4 kilobytes. This would mean that during a checkpoint there would be 3072 files in the checkpoint directory (1536 files for the old checkpoint and 1536 files for the new checkpoint). Each file requires an *inode*, or special disk block, that provides a map of the file's data. Inodes are a limited resource in the UNIX file system, since only a fixed number exist. This fact, coupled with the fact that directory searches are linear, make 3072 files, or even 1536, impractical. Performance of operations on a directory with so many files would become severely degraded and inodes would become a scarce commodity.

To make disk based checkpointing practical, pages are divided up among a fixed number of files. In the system implemented, a process' pages are managed as a hash table. When a page is needed, the hashing function is applied to locate the appropriate bucket, and a linked list is traversed to find the requested page. This hash table maps perfectly to the file limit solution. For example, if the hash table has 25 entries, a process will create, at most, 25 files in a memory checkpoint. All the pages in a bucket are written to the bucket's page file. The number of files is then a function of the number of processes in the application, rather than the amount of memory used. If there are 10 processes, at most 250 files will be created.

When a page is modified, the entire list of pages in the modified page's hash bucket must be checkpointed again. This indicates that it would be advantageous to put pages that are spatially close into the same bucket. Then if an application touches only a small localized set of pages between checkpoints, the number of buckets that will have to be checkpointed will be minimized. The hashing function that achieves this is:

$$\textit{hashbucket} = (\textit{pagenumber}/\text{LOCALPAGES}) \bmod \text{NUMBUCKETS}$$

where the LOCALPAGES is the number of pages grouped together and NUMBUCKETS are the number of buckets in the hash table. If LOCALPAGES were 10, for instance, page 1would map to bucket 0, as would pages 2-9. Page 10 would then map to bucket 1. Figure 34 shows an example hash table and the layout.



Hash Function:

(pageno/10) mod 25

Figure 34. Example hash table

In addition to placing the pages on disk in easily identified files, it is also necessary to know which files belong with which checkpoint. Since incremental checkpointing is used a page file from the first checkpoint may be the current file even after 6 checkpoints. If a failure occurred during a checkpoint, the recovery procedure would have to perform a search of the directory to find the most recent checkpoint for each hash table entry. This is avoided through the use of a *process page catalog*. When a checkpoint is taken this file is written out for each process (e.g. *page-cat.432.4*). The catalog has an entry for each bucket in the hash table that identifies the most recent checkpoint of that particular bucket. For example, if a process touches a page in hash table entry 0 before the first checkpoint and then doesn't touch it again, the 0th entry in its page catalog would be 1.

3.5.3 Disk Checkpointing

The mechanism that performs disk checkpointing works in conjunction with the journal disk recovery algorithm. At checkpoints, all information that has been saved by the journaling policy is deleted, and all data structures that monitor the data that was saved are cleared. The size of files that are open are noted and this is then used as the initial size when rolling back to the checkpoint.

To recover the disk to a checkpoint, the same procedures are followed as in the journal recovery policy.

3.5.4 Operating System Checkpointing

When a process is checkpointed, operating system data structures such as the process control block, communications structures and file descriptors opened by the process, are saved to a file. This file contains the "brains" of the process. The only issue that needs to be addressed is resource identifiers during a recovery. Many operating system data structures contain pointers to other structures. When a process is allocating data structures during the recovery it is extremely unlikely that it will get the same data structures it had in the original run. For this reason a fixup data structure is used to record the location of pointer references that cannot be resolved until after all processes have been recovered.

A visible example is the application's family tree. Each process control block contains pointers to the process control blocks of the process' parent, child, older sibling and younger sibling. As the recovery proceeds the mapping of old process identifier to new process control block is recorded in the fixup data structure. Instead of storing process control block pointers in the family tree entries, the old process identifiers are recorded. As one of the final phases of the recovery, a fixup routine goes through each process recovered and changes the family tree pointers by looking up the old process identifiers in the fixup mapping table and replacing them with the new process control block pointers. This same process occurs for other less visible structures such as terminal and communications control data structures.

3.5.5 Taking A Snapshot

Taking a snapshot of the application involves obtaining a view of each process that is consistent with an overall view of the application (consistency is covered in greater detail in Chapter 4). Basically, this means that if one process was checkpointed with a state indicating it received a message from another process, the other process' checkpointed state must reflect the fact that it has sent the message. A second factor that must be addressed is that all of the processes must be in *checkpointable* states. A checkpointable state means that a process is in a state that is self contained and can be readily reproduced during a recovery.

When the checkpoint routine looks at a process to see if it can be checkpointed the process can be in a variety of different states:

4. executing in its own address space
5. in the entry or exit of a system call
6. temporarily blocked in a system call
7. semi-permanently blocked in a system call

The first case is a trivial one to handle. The process must be frozen and a snapshot of its registers, operating system data structures and address space must be made. To recover the process a new process must be created and loaded with the saved data.

The second case is not as easy to deal with. When the process is executing a system call, its registers and address space reflect the fact that it is waiting for the call to return. The process' operating system data structures might be in the process of changing or about to change depending on the operations of the system call. To recover a process in this state the operating system thread that is servicing the system call must also be recovered. While this might seem easy, it actually introduces a great deal of additional complexity that should be avoided. It is therefore desirable to have the process move out of this state and into another one that can be checkpointed such as the first case.

The third introduces additional levels of complexity. A process that made a system call to read data from a file might be waiting for the call to return. The operating system thread servicing the call could be temporarily blocked while it is waiting for the disk to provide the requested data. The state of the process is therefore spread across its user space, the operating system, and the device involved in the system call (in this case the disk drive). It is impractical, or in this case impossible, to checkpoint the dynamic characteristics of the device. If the disk drive were to be checkpointed, all the device driver routines would need to be checkpointed and the recovery routine would have to be able to reset the disk to be reading the exact byte it was reading at the time of the checkpoint.

The last case is similar to the previous one. A process will be in a semi-permanent sleep if it has made a blocking request for data from a communications port and the port is empty. The process will wait indefinitely for data. This case can be detected by looking for the process on the operating systems semi-permanent sleep queues. Again, checkpointing the process would mean that the operating system would need to be checkpointed.

So, how are the dilemmas of cases 2-4 to be solved? The first key to the solution is to realize that if a process is in a semi-permanent sleep (4), and the system call that lead to the sleep was restarted given the same conditions, that the process would again enter the sleep. Next, processes in cases 2 and 4 will, within a bounded period of time (provided there is a fair scheduling algorithm) move into either state 1 or state 4. Of course there is no guarantee that if the process is polled that it will not always be in states 2 or 3, just executing different system calls. To prevent this from occurring a flag is set in the checkpointing entry sentry to force processes to halt before entering a system call if a snapshot is being taken. A process blocked in an entry is a variant of state 4.

If a process is in a state where the system call it is in needs to be restarted in the recovery, the registers must be fixed so that this will occur. This means that the process program counter and stack pointer must be copied and adjusted before they are saved.

Once a snapshot has been taken, individual checkpointing threads are spawned that write out the memory checkpoint of each process. The hash table queues are written to files and asynchronously flushed to speed their movement to disk. After all processes have been written to disk, the checkpoint is committed by having the last thread first write the new checkpoint number to a *committed checkpoint log* and then perform a synchronous flush of all disk buffers. This insures that the new checkpoint is on disk and that the disk checkpoint associated with the new checkpoint is also stable. The checkpoint is now said to be *committed*. The last thread then deletes the old checkpoint, journal, and disk checkpoint information. Performance of the application during this commit period can be affected because of the additional CPU load caused by the threads performing the commit and the load on the disk caused by writing the checkpoint.

A high level pseudocode listing of the snapshot algorithm is shown in Figure 35. The first step of the algorithm is to suspend all processes involved with the application. A loop is then entered where each process is checked to see if it is in a safe state to checkpoint. If a process cannot be checkpointed, it is unsuspended so that it can move into a safe state. After a each process has been checked, the snapshot algorithm sleeps for a small time interval if there were any processes that were not checkpointed. The sleep interval should be large enough so that on average, any processes not in a checkpointable state will enter one within the time interval.

```
routine snapshot()
   begin /* snapshot */
   suspend all processes in the application;
   set flag indicating processes should stop in system call
         entries;
   while(processes left not checkpointed)
         begin /* while */
         for(each process not checkpointed)
               begin /* for */
               curproc := next process not checkpointed;
               /* state 1 */
               if(curproc is executing in its own address space)
                     begin /* if */
                     checkpoint_process(curproc);
                     mark curproc as checkpointed;
                     end /* if */
               else
                     begin /* else */
                     /* state 4 or call entry */
                     if(curproc is in system call entry or in a
                           semi-permanent sleep)
                           begin /* if */
                           copy curproc's regs and
                                 adjust so system
                                 call will be restarted
                                 in recovery;
                           checkpoint_process(curproc);
                           mark curproc as checkpointed;
                           end /* if */
                     else
                           begin /* else */
                           /* process not currently
                                 checkpointable
                                 (state 2 or 3) */
                           resume process;
                           end /* else */
                     end /* else */
               end /* for */
         if(all processes not checkpointed)
               delay();
         suspend processes that were resumed above;
         end /* while */
         resume all processes;
   end /* snapshot */
```

Figure 35. Snapshot algorithm

3.5.6 Device Initialization

During application execution the state of external devices may be changed. An example is a video display card which is set to a certain mode of operation. If no provision is made for these operations, then recovery to a checkpoint may result in the device being in a state that is inconsistent with the application's expectations. There are two ways to checkpoint a device's state. The method of choice is determined by the characteristics of the device. If the application's modifications of the device's state are transparent to the operating system (as in the case of a memory mapped video card), some mechanism must exist for reading the entire state of the device from the operating system. To recover the device its entire state must be read from the checkpoint and then used for its reinitialization.

If all state modifications are visible to the operating system through system calls, it may be advantageous to log state modifying commands to a special device checkpoint log. When a device is being recovered, state modifying commands must be replayed in succession, to move the device to the correct state. Unlike other checkpoint files, the device checkpoint log is not deleted when a new checkpoint is taken, but is copied to the new checkpoint and appended with any state modifying commands that have occurred since the previous checkpoint. If the number of modifying commands is expected to be high for a particular device resulting in a checkpoint log that is the same order of magnitude in size as the device's state, then the option of checkpointing the device's state in its entirety should be pursued if possible.

In the system implemented there are examples of both types of device. The video display card contains part of the X application's state - namely, the display image. X sets the device into a graphics mode, initializes a palette of colors for display, and then draws the display image in the display card's memory. These state changes are all made through special assembly language device instructions and through memory mapped I/O, and therefore are transparent to the operating system. Even if they were not transparent, the number of state modifications made when a window is drawn on the screen, for example, makes logging impractical for this type of device.

The second type of device is exemplified by the mouse and the keyboard. In the Mach/UX architecture, the low level device driver's are in the kernel and since the checkpointing policy has been placed in the UX server, it does not have access to the state of the device drivers. During the course of X initialization, `ioctl` (I/O control) system calls are made to set the state of the mouse and keyboard device drivers. The checkpointing policy journals these calls into a *device log file* so that at the beginning of a recovery the device drivers can be initialized by replaying them.

3.5.7 Checkpoint Interval

Checkpointing implies a transactional saving of state to the stable medium. Because this is usually an expensive operation, checkpoints must be spaced in time. The issues involved in determining the checkpoint interval include the cost of checkpoint, limits on the size of the journal, and recovery time limits. Each factor is influenced by both the implementation and by application requirements. The checkpoint control mechanism should be flexible allowing for the interval to be set at run-time with some of the interval specification options being:

- Time interval based
- Journal size-limit based
- Application or user directed
- Event determined

Measurements of checkpoint overhead will provide user's guidelines that can be followed to achieve specific application needs.

3.5.8 Checkpoint Recovery

Recovery of a checkpoint is a fairly straight-forward process. First, the commit log is read to ascertain the most recently committed checkpoint. A check is then made to see if an uncommitted checkpoint exists, and, if so, it is deleted. An uncommitted checkpoint means that there is a second journal and disk checkpoint that must be integrated with the journal and disk checkpoint of the committed checkpoint. The committed and uncommitted disk checkpoint and journal files are then spliced together. In the case of the journal, this means appending the second journal to the first. For the disk checkpoint a process whereby the file identifiers of the second disk checkpoint are changed to corresponding id's of the first checkpoint or assigned new id's must be performed.

Processes are created for each process in the checkpoint with their process control blocks, registers, and memory map restored from information in their individual checkpoint files. It is in this part of the recovery that the fixup structure referred to earlier is created. Next, the memory maps are filled with the checkpointed pages. External pager hash tables are recreated as this occurs. Disk checkpoint recovery is then performed followed by the fixup routine that adjusts pointers. Finally, all the processes are started. These steps are outlined in Figure 36.

3.6 Checkpointing Examples and Performance

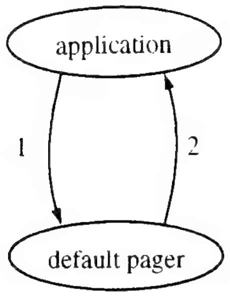
This section presents measurements of overheads for the external pager, taking a snapshot, and committing a checkpoint. Just as for disk checkpointing, it is necessary to measure instantaneous overheads and profile real applications to obtain an overall view of the checkpointing performance impacts. The first part of this section presents measurements based on a well defined synthetic program. This is followed by examples of real checkpoint sessions taken on the same applications that were used in the journaling measurements.

```
routine checkpoint_recover()
   begin /* checkpoint recover */
   read the commit log to obtain the committed checkpoint
         numer;
   if(an uncommitted checkpoint exists)
         begin /* if */
         delete the uncommitted checkpoint;
         append the uncommitted journal to the commited journal;
         integrate the uncommited disk checkpoint with
               the commited disk checkpoint;
         end /* if */
   for(each process of the application)
         begin /* for */
         create a new process;
         restore the process control block from the checkpoint;
         restore the address space of the process;
         end /* for */
   disk_restore();
   do any pointer and resource id fixups;
   start all processes;
   end /* checkpoint recover */
```

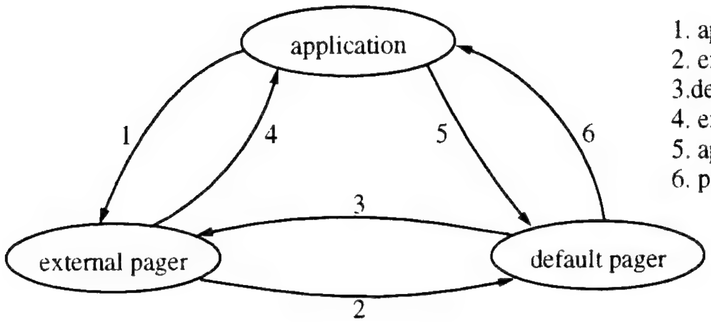
Figure 36. Checkpoint recovery algorithm

3.6.1 External Pager Overhead

Out of all the factors affecting the performance of an application that is using the checkpointing and journaling policies, the use of the Mach external pager facility is the most detrimental. There are two main reasons for this: the first is that the external pager adds a level of indirection in the page allocation path and the second is that each page that is backed by the external pager is also backed by the kernel pager. Request paths used when an application is backed by the kernel's default pager are shown in Figure 37(a). Requests generated when an external pager is used are shown in Figure 37(b). When the application is running and takes a page fault on a new page (initializing data for example), the kernel sends a request for the new page to the external pager. The external pager must itself allocate a page to provide. This generates a page fault in the kernel that causes a request to be sent to the kernel's default pager for the new page. The default pager supplies the page to the external pager which, in turn, supplies the kernel with the page. When the page is modified by the application, the kernel makes a copy of the page to dedicate to the application. Therefore, two memory pages are dedicated to backing one application page. This results in extra load on the physical memory system. The use of more pages means that more pages must be swapped into and out of physical memory. Each pageout means a disk write as the page is written to the paging file for temporary storage.



(a) page fault with kernel pager


application
external pager
default pager
1
2
3
4
5
6
1. application requests page
2. external pager requests page
3.default pager supplies page
4. external pager passes page
5. application copy faults on page
6. pager provides private copy

(b) page fault with external pager

Figure 37. Page fault paths

To measure this overhead a synthetic program was written that has a global data structure. The program touches each page of the data structure once by writing a value to the first byte of each page. The size of the data structure is varied by modifying the program and recompiling it. Time per page access was measured by taking the total time and dividing by the number of pages touched.

Results of running the program with the external pager and the default kernel pager (native) are shown in Figure 38. The small values on the left side of both plots indicate that the amount of virtual memory accessed by the program, the kernel and the external pager (if used) fits within the physical memory allocated for swapping.In the plot of the external pager run a dramatic jump occurs at 3 MB. This indicates that swapping has begun, resulting in pages being stored in the disk paging file. Page access time then goes up because to touch new pages the kernel pager must write pages to disk to make room for the new pages in physical memory. This effect is seen at about 6 MB when the program is run with the kernel pager as its backing memory manager. The jump is much higher in the external pager case because when the external pager is used there is more than one page allocated for each page of program data that is touched.

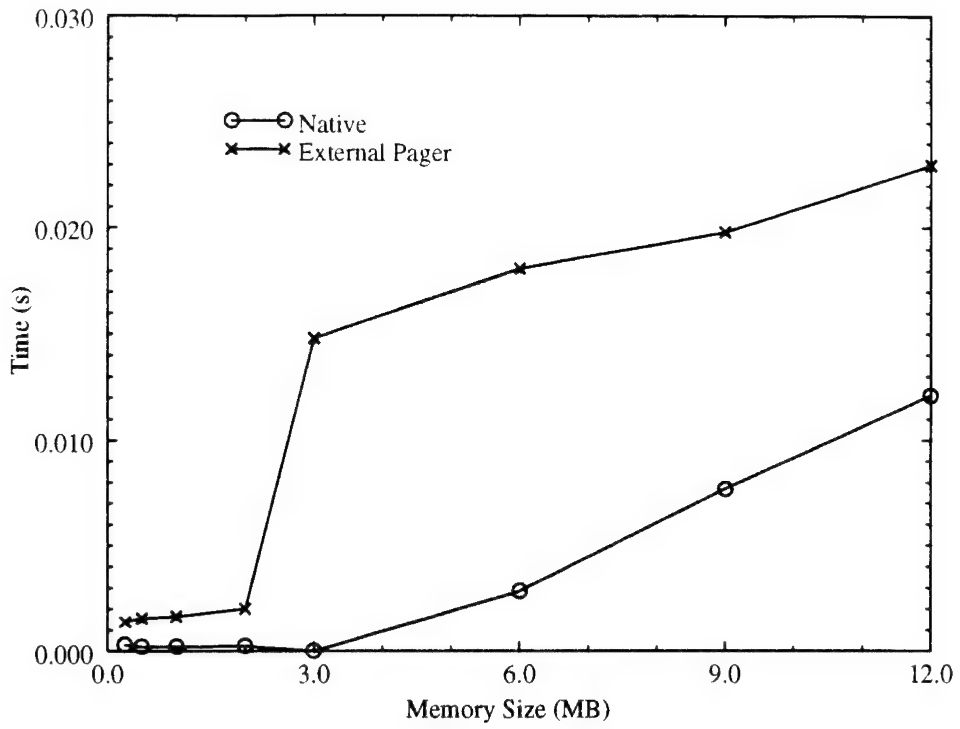


Figure 38. Time per page access for default and external pager

More information as to what is occurring in Figure 38 can be obtained by plotting the number of page faults that occur in each experiment. The number of total page faults are plotted in Figure 39. External pager runs generated 2-3 times the number of page faults as default pager runs. Figure 40 shows the number of pageins and pageouts generated. If this is compared with Figure 38, the sharp rise in page access times correlates with the beginning of the pagein curves. Pageins are synchronous events since the application cannot continue until the page data is available. Page-outs work by a different mechanism since unmodified pages that are paged-out do not need to be written to disk.

It is clear that the use of an external pager the memory requirements of an application by two to three times, with adverse effects on application performance. The additional memory demands could be eliminated if the kernel default pager were instrumented with sentry points.This would allow for the checkpointing policy to work directly with data structures and pages that are backing the application. To avoid delays that would have been involved in modifying the Mach kernel, this feature was not incorporated in this initial work, but it can be readily added in future versions.

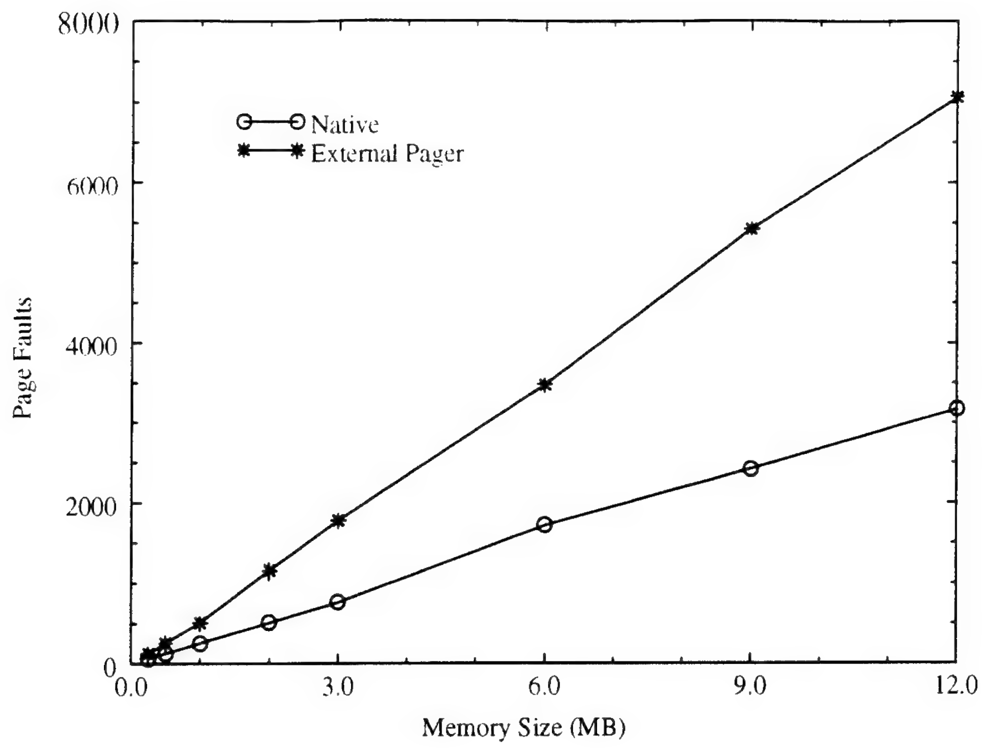


Figure 39. Page faults generated by memory touch

3.6.2 Snapshot Performance

The most visible part of a checkpoint to a user is the snapshot. During a snapshot the application is essentially suspended so the snapshot's duration is an important factor. A smaller snapshot would in most cases allow for more frequent checkpointing. To measure the performance of snapshots in a controlled environment, the synthetic program used in the external pager profiles was used again. The program's global data size was again varied and each page was touched. Checkpoints were then taken and the snapshot intervals measured. Measurement results are plotted in Figure 41.

The duration of each snapshot increases at roughly a constant factor as the amount of memory used increases. Recall that all that is done during this phase is that the snapshot algorithm is writing the process control block of the program and its registers to a file while requesting that the kernel flush all of the application's modified pages. When the kernel notifies the snapshot algorithm that a page has been flushed it does not need to immediately provide the page. It simply marks the page as flushed so that if the application modifies the page again it will have to request it from the external pager. The page is provided to the checkpoint algorithm when it touches the page to save it to the checkpoint file. When there is only one process being checkpointed, this

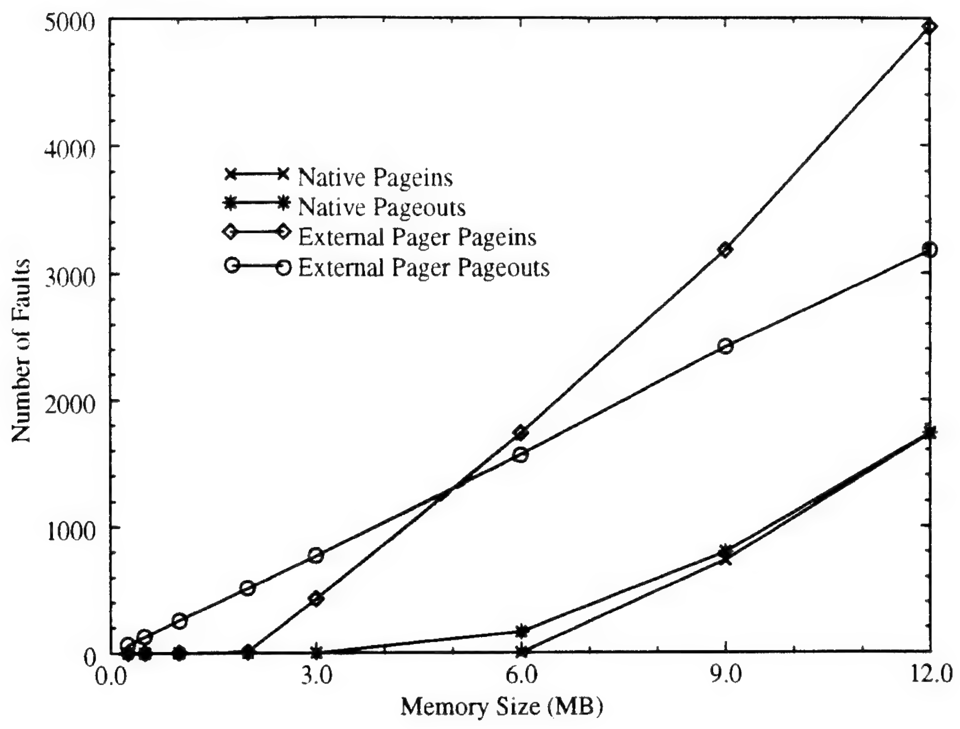


Figure 40. Pageins and pageouts for memory touch

occurs during the commit phase of the checkpoint. If the kernel provided the page when the flush request was made, in many cases it would have to pagein the page from the disk paging file substantially increasing the snapshot duration.

3.6.3 Commit Performance

The time period between completion of a snapshot and a checkpoint is committed is the *commit delay*. During this time the application is resumed from its suspended state, but it may experience performance degradation due to the load of checkpoint commit routines on the processor and the disk. During this period, the kernel is flushing pages to the external pager, and the checkpointing threads are writing the pages out to disk. To measure the commit delay for various sizes of checkpoint, the synthetic program was again used. Two curves are graphed in Figure 42, one for compressed checkpointing, and one for an uncompressed checkpointing. Run-length compression usually is able to significantly reduce the space taken by memory pages. It is therefore used in the implementation to compress both checkpointed pages as well as video data. In addition to saving disk space, compression can improve performance because fewer disk blocks need to be written and flushed to disk. The compressed curve is a lower limit on the commit duration because only

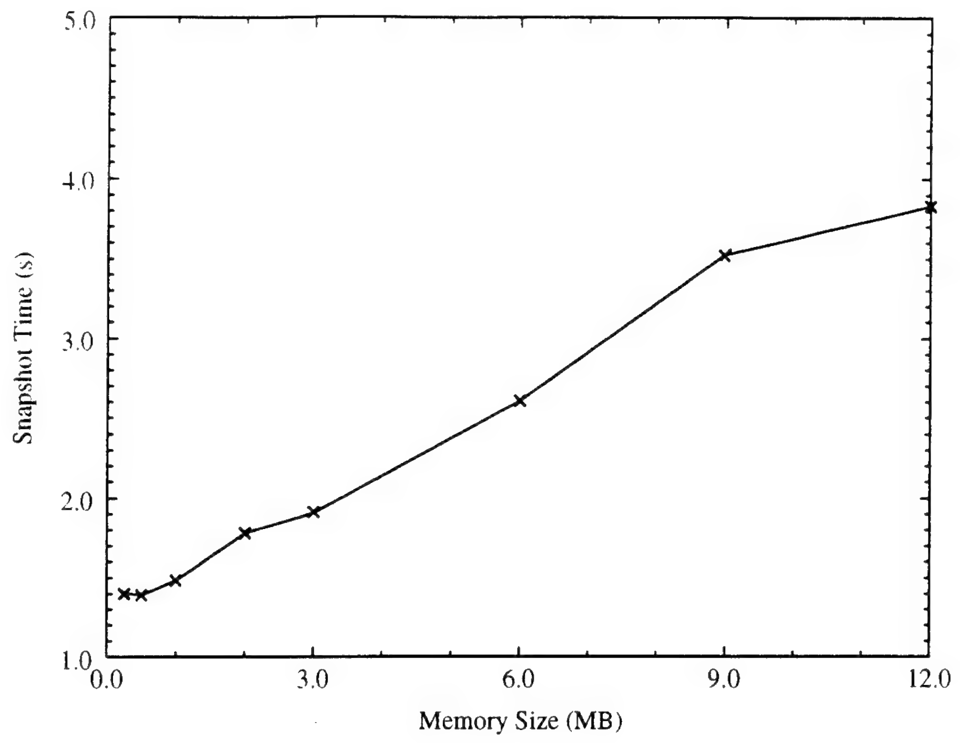


Figure 41. Snapshot duration for synthetic program

one byte of each page is nonzero (the byte that is modified). This is the optimal situation for the compression algorithm and results in the smallest checkpoint size and therefore the lower bound on commit delay. The uncompressed plot represents a checkpoint where every byte on the checkpointed pages differs from its neighbor so that no run-length compression is possible. It therefore represents the upper bound on checkpoint size and performance. As in the memory touch experiments, increases in paging activity starts to create disk accesses causing the kinks on the left side of the graph.

For 12 megabytes of data, one can expect a commit time of 2 to 2 and 1/2 minutes. Recall that the application is allowed to run during this time. The effect of performance degradation during this interval can decrease system responsiveness, a factor that needs to be considered in selecting the checkpoint interval.

3.6.4 Recovery Time

Although most people would consider recovery time of secondary importance if the alternative was the loss of critical data, it is desirable to minimize it. Figure 43 shows the time taken to recover the synthetic program. *Recovery* here refers to the time that it takes to recreate a process

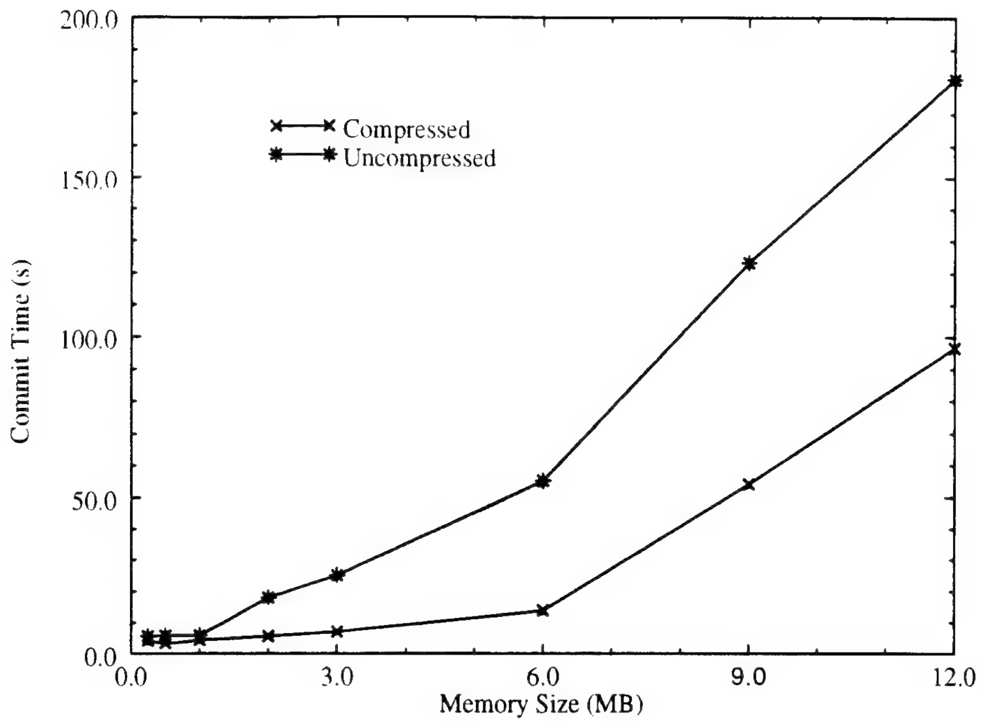


Figure 42. Checkpoint commit delay

from a checkpoint. Comparison of Figure 42 and Figure 43 shows that recovery time for a given data size are about one third commit time for the same amount of memory. Much of this is due to the fact that the checkpoints are read instead of written and disk read operations are faster than write operations. Also, since there is no memory copying or flushing in the recovery, there are fewer page faults. Once again, the sharp rise coincides with disk paging activity.

3.6.5 Examples of Checkpointing

The dilemma facing checkpoint measurement is the same that confronted journaling measurement: there are no standard benchmarks. Synthetic programs like the memory toucher have very little relevance when the question is asked, "*how long will a checkpoint of this application take?*" Results in the synthetic case can only be used to make crude approximations of the performance of real applications. Even when a real application is used in measurements, the correlation between the circumstances that existed at the time of the checkpoint (state of the paging file, buffer cache, etc.) can be significantly different than for the same application just a few seconds later. Still, measurements must be taken and presented if only to provide an order-of-magnitude reference.

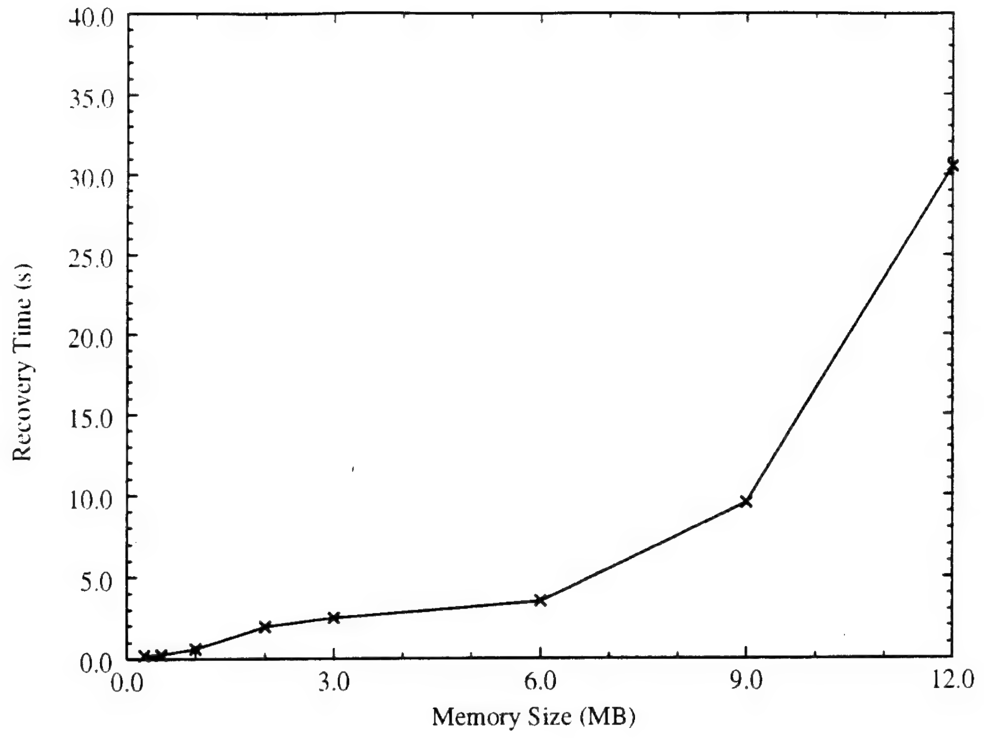


Figure 43. Recovery time

A few programs were selected as representative of the most popular generally available programs used in the X system. `Bitmap` is the drawing program shown in Figure 29. With it, a bitmap can be drawn or edited, and saved or read from a file. A variety of drawing functions are presented as buttons along the left side of the window. The `gnu-emacs` editor is perhaps the most popular X/UNIX text editor and is commonly used in program development (like the development of Mach and the sentry policies). It has powerful facilities that allow one to view and edit multiple files simultaneously. `Xterm` gives terminal programs like shell and cshell (`csh`) an interface to the X-window system. With its use the user can have several shell programs running concurrently and moving between them is as easy as moving the mouse cursor from one `xterm` to another. `X-standard` is the name given to X and its batch of standard startup clients. In these experiments, as in the journaling runs, an `xterm`, window manager (`twm`), and a clock (`xclock`) are all part of the environment. Finally, `X-large` is a more typical working environment. It includes all that `X-standard` does, plus 3 additional `x-terms` and a `gnu-emacs` editor with a file loaded.

The first set of measurements in Table 5 show checkpoint performance immediately after the programs have finished their startup sequences. For bitmap this means that a `bitmap` editor window has been drawn on the screen and is ready for input. The same applies to `gnu-emacs` and `xterm/csh`. `X-standard` initialization is complete when the `xclock` and `xterm` windows are drawn and the shell (`csh`) in the `xterm` is ready to accept input. `X-large` is checkpointed after all 4 `xterms` have been created and the `gnu-emacs` editor has finished its initialization after loading a file.

Application	program size(byte)	pages ckpted	page faults	snapshot delay(s)	commit delay(s)	ckpt size(b)	% of no cmprsn
bitmap	559K	73	179	1.6	1.9	137K	47
xterm+csh	813K	103	177	1.8	2.2	155K	38
gnu-emacs	1466K	58	159	1.9	2.1	149K	64
X-standard	1095K	441	1422	8.3	8.5	591K	33
(video card)						29K	3
X-large	9881K	826	1661	20.7	8.4	1236K	37
(video card)						60K	6

Table 5. First checkpoint performance

A variety of different metrics are shown for each program with all sizes in bytes and all times in seconds. Program size indicates the size of the object file, or files, and can be an indicator of the complexity of the program. Checkpoint size is the size of the memory checkpoint, which is compressed. The final column shows the amount of compression that was possible by comparing it to the projected uncompressed size. The video card is checkpointed when X is checkpointed.

If no data compression were applied to the video data, its checkpoint would be 1 MB in size. Fortunately, video, especially an X desktop image, tends to high compression ratios when a run-length algorithm is applied. In this case the video checkpoint is 3-6% of the original size. The table shows that even for very large applications (`X-large`) the snapshot is only 20 seconds in duration. A delay such as this is usually tolerable in an interactive environment about once every 30 minutes. The total checkpoint sizes are less than 2 MB in size. If this is combined with 30 minutes of journal, which is about 3 MB and an additional checkpoint (before a new checkpoint is committed both the old and the new checkpoint exist on disk) the disk space requirements of a such a system are about 10 MB.

To see the effects of data compression on checkpoint duration, measurements were taken for checkpoints of the `X-standard` and `X-large` applications without data compression. Measurements obtained are shown in Table 6. Times are shown for snapshot delays, and commit delays, with the sum of those delays shown in the checkpoint duration column. Total durations are reduced by about 30% in both cases when data compression is used. This savings is due to the fact that fewer disk blocks must be written when the data is compressed to take less space. Also of interest in the table is the fact that the commit delay is much less than the snapshot delay for `X-large` when no compression is used. When a snapshot is taken of a process, a thread is immediately spawned to being committing that process to disk. This means that in multi-process applications, the commit of one process may overlap the snapshot of another process. In this case, most of the processes had finished committing their checkpoints before the final processes had finished their snapshots.

Application	snapshot delay (s)		commit delay (s)		checkpoint duration (s)	
	cmprss	none	cmprss	none	cmprss	none
X-standard	8.3	14.6	8.5	10.5	16.8	25.1
X-large	20.7	35.3	8.4	8.5	29.1	43.8

Table 6. Checkpoint duration with and without compression

Checkpointing an application after it has finished its startup indicates how much data is modified during initialization. Measurements for a second checkpoint are shown in Table 7. The checkpoints were taken after the programs were exercised in their respective ways. `Xterms` commands were entered, a drawing was entered with `Bitmap` and text was edited with `gnu-emacs`. The table shows that the amount of data saved in a second checkpoint can be significantly smaller than in a first checkpoint. As expected, snapshot durations are also smaller.

Application	pages ckpted	page faults	snapshot delay (s)	commit delay (s)	ckpt written(b)	% of no cmprsn
bitmap	53	93	1.4	1.6	91K	42
xterm+csh	102	159	1.3	2.2	149K	36
gnu-emacs	63	132	1.4	2.1	164K	65
X-standard	266	1154	5.7	6.0	436K	41
(video card)					35K	3
X-large	356	1238	11.2	3.7	595K	42
(video card)					65K	6

Table 7. Second checkpoint performance

The final measurement of recovery time is shown in Table 8. Only `X-standard` and `X-large`

Application	Recovery Time (s)
X-standard	3.2
X-large	16.9

Table 8. Checkpoint recovery

are shown because the other programs are not true applications. If considered individually, a program like `xterm` violates the system model because it communicates with processes that are not descendents of its root process. X must therefore be the root process for any X based program.

Recovery times are comparable to snapshot durations, and are relatively low considering the complexity of a full recovery of the scale of `X-large`, for example. Low recovery times such as these indicate that user's should expect checkpoint recovery to be under one minute in duration. Using the example parameter of a 30 minute checkpoint period, a user could expect a total average recovery time of about 5-10 minutes. This is based on 50% journal recovery compression for an average of 15 minutes elapsed time since the last checkpoint. As the duration of the application session increases this number results in increasing savings in both time and effort of reproducing a lost session. Even if little data is lost, the low recovery latency makes checkpoint recovery practical.

3.7 Summary

This chapter has presented the design and implementation of practical and efficient single computer fault tolerance through the use of journaling and checkpointing. The techniques used and demonstrated the versatility of the sentry mechanism presented in Chapter 2.

A wide range of challenging issues were addressed during the development of these policies leading to the development of new algorithms that, working together, support the recovery of concurrent, communicating processes running on a single node. Contributions of special significance related to the journaling policy are the journal replay, signal replay and disk checkpointing algorithms. The journaling techniques make possible the recovery through replay of today's workstation benchmark of application size and complexity, the X-window system and its clients. Signal replay presents a novel approach to handling the recovery of signals in the UNIX 4.3 BSD environment.

The disk checkpointing algorithm is a new contribution to recovering persistent storage, an issue usually ignored in previous recovery work. Because the disk checkpointing algorithm is based on the file as the unit of recovery, rather than the logical disk block as in previous algorithms, the new algorithm requires no modification of the file system and is easily integrated into the sentry framework. The combination of these algorithms provide for less than 10% overhead in fault free execution for the majority of typical workstation software with at most one keyboard or two mouse inputs being lost in a failure. Recovery usually provides replay compression of between 5 and 15 meaning that time, if not data, is saved in the case of a failure. Journal data generation rate is on the order of less than 100 KB per minute indicating that a 10 MB disk partition, considered small by today's standards, is normally enough to store one hour of application behavior.

The checkpointing policy led to the design of a snapshot algorithm for concurrent processes that are making system calls. This is new in that existing algorithms do not address the challenge of checkpointing processes that may be blocked in the operating system or performing device accesses. An organization strategy is also presented that solves the problem of file explosion in a disk based checkpoint system. Measured snapshot durations of on the order of less than one minute for memory intensive applications and checkpoint sizes of less than 2 MB represent exciting results that prove that a hard disk is a suitable alternative to other forms of stable storage. If checkpointing is performed every 30 minutes, less than 10 MB of disk storage in total is needed for both journaling and checkpointing, and recovery latency will be on average less than 5 minutes (on average a crash will result in 15 minutes of journal - 30/2 - which is replayed in 5 minutes).

The next chapter describes research into computing environments consisting of multiple computers. These environments have been divided into the two areas of mirrored systems and distributed systems. Issues related to fault tolerance are discussed with respect to the areas and a number of new algorithms are developed that are projected to have very low average overheads. Sections that follow discuss the integration of these algorithms with the sentry mechanism.

Chapter 4

Multiple Computer Fault Tolerance

4.1 Introduction

The algorithms introduced in Chapter 3 provide fault tolerance for single computer, concurrent applications. While a large class of applications fit this model, there are an increasing number of applications that cannot tolerate the outage caused by the recovery procedure of rebooting and restarting or that consist of distributed computation. There is also an increasing trend toward distributed computing, where processes on physically separate computers cooperate through message passing. Additional computers are necessary to effectively deal with these additional requirements, and this chapter focuses on sentry based software solutions that use multiple computers to address them.

There are two ways that multiple computers can be used to provide fault tolerance: either through mirrored redundancy or distributed redundancy. In mirrored redundancy, multiple computers are made to work in lock step with one another either through hardware or software techniques. The computers use voting and other fault detection techniques to determine if a failure has occurred and mask it from the outside world. An image of one highly available computer is presented to the rest of the system. All major vendors of fault tolerant computers use some form of hardware redundancy and lockstep to provide a view of a continuously available computer [33]. Levels of high availability are necessary in computers that act as file or compute servers for a network of client computers or that are being used in critical computation such as an on-line reservation system or a transaction data base. A software approach to mirrored systems, however, has many potential advantages over the existing hardware approaches including lower cost, the ability to use current hardware, low maintenance, and ease of installation. This is the area explored in the first part of this chapter. Design of a high performance mirrored system that is based on the proven technology of Chapter 3 is presented. The solutions are based on synchronizing computers at the highest possible level while maintaining the guarantees of availability that are made by much more costly hardware techniques.

Multiple computers are used for distributed redundancy in the myriad of distributed recovery algorithms that have developed over the past decade. A distributed computation is an application that has processes physically distributed among a set of networked computers. Fault tolerance is achieved with checkpointing and journaling, but different variations on this theme have different levels of fault tolerance, performance, types of overhead, and underlying assumptions. Initially this work began as an investigation into the suitability of the sentry mechanism as the framework for these algorithms, but in the end a new type of distributed recovery algorithm was introduced that, like the mirrored system technology, is based on the tested ideas of Chapter 3. Specifically, the new distributed journaling algorithm only saves information about message ordering, rather than message content. This contribution is a direct extension of the sequenced journaling used in the single computer fault tolerant algorithms. In addition, a new type of loosely synchronized checkpointing is introduced to work with the journaling algorithm. Together, these algorithms can provide increased performance with reduced complexity over existing algorithms.

Neither of these new ideas has been implemented to date, however, the fact that they are based on the single computer technology makes their designs credible.

4.2 Mirrored Systems

This section presents algorithms for two and three node mirrored computer systems. The first part of the section describes characteristics of a software based mirrored system (SBMS) and briefly mentions the issues involved in an implementation. This is followed by an examination of each implementation issue in depth and the section is concluded with a comparison of SBMSs to existing fault tolerant systems. In this section the words computer, node, and processor will be used interchangeably to refer to a complete computer system including a CPU, main storage, and secondary storage.

4.2.1 System Description

The goals of a SBMS are to provide the following over a single node workstation:

- permanent failure fault tolerance
- enhanced fault detection
- continuous availability

The single computer techniques of Chapter 3 can only tolerate transient hardware and software faults and certain permanent failures. A two processor SBMS is able to tolerate one permanent fault and a three processor configuration can tolerate up to two concurrent permanent faults. In addition, the nodes in a multi-computer configuration can use each other as reference points to perform fault detection. Finally, a node that has failed can be replaced with a new node with minimal disruption to system service.

An SBMS system would consist of the following (parenthesis are used for the three node system) components:

- 2 (3) off-the-shelf PCs or workstations
- Floppy disks containing an SBMS initialization system
- 4 (6) ethernet boards
- 2 (3) ethernet cable
- 2 (3) terminals for monitoring and system control

An architectural picture of a two node system is shown in Figure 45 and Figure 45 shows a two processor configuration. The computers are connected to a network via ethernet, and there is also an ethernet cable serving as a direct link between the nodes. For all intents and purposes the network the system resides on would see one computer instead of two. The exact method whereby the two nodes would receive the same input is an implementation detail dependent on the network. One possibility is to assign the same network addresses to the two nodes. Another alternative is to have one node pass the input to the other node. To reduce network load, one of the mirrored computers would consider itself the output node.

4.2.2 Issues

There are three issues which must be addressed in mirrored system fault tolerance:

1. synchronization
2. fault detection
3. integration

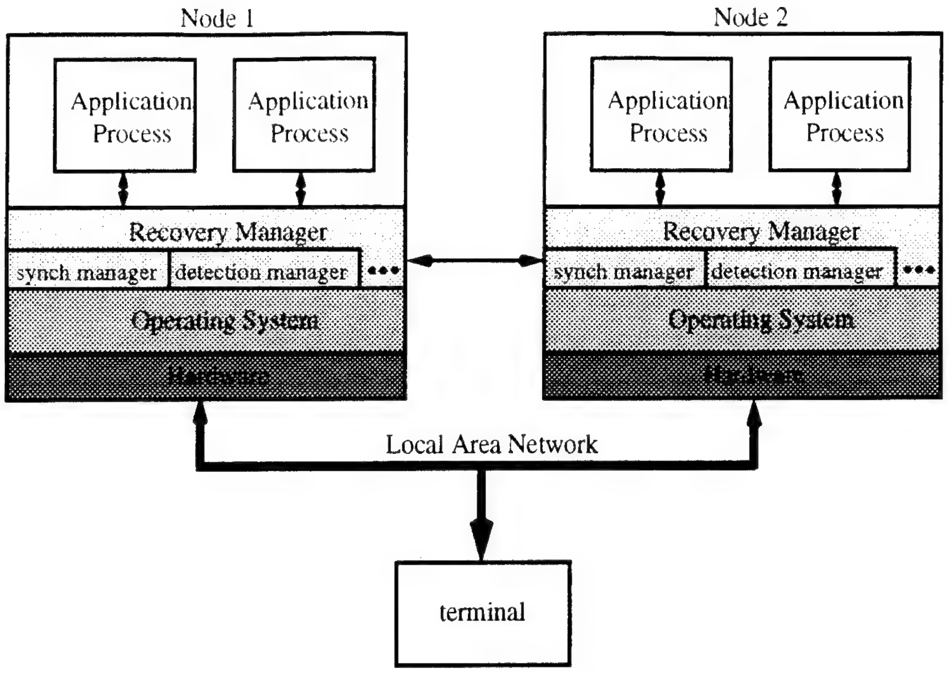


Figure 44. Architecture of two processor mirrored system

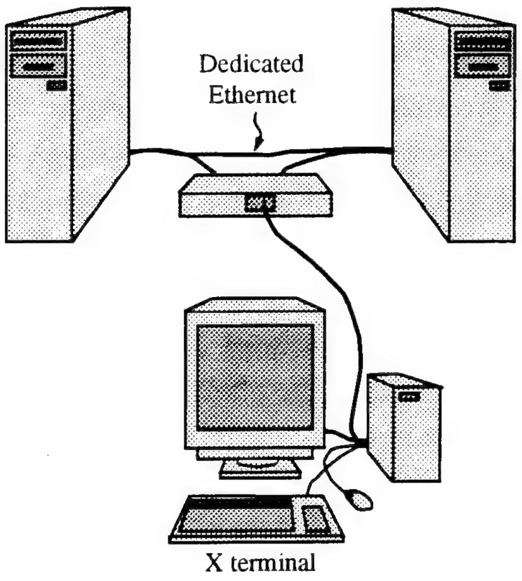


Figure 45. Two processor mirrored system

During normal execution each machine must be kept synchronized with the other machines in the system. This synchronization ensures that the machines are essentially clones of one another and that they are performing the same operations at approximately the same time. If the system is properly synchronized it will not be biased to any particular machine remaining fault-free.

In order that recovery can take place, faults in the system must be detected and handled. The native machine fault detection mechanisms will catch the majority of faults, but the fact that the machines are synchronized provides opportunities for enhanced detection mechanisms. If the machines diverge (lose synchronization), a fault has occurred. A further level of detection compares outputs and system call results across machines for agreement.

Finally, when a node fails and a new node is brought in for integration into the system it is necessary to:

- make the new node an exact replica of the existing nodes
- minimize performance degradation of the functioning nodes
- minimize integration latency

4.2.3 Baseline Node

The nodes that will be considered here for analysis are ones that are representative of typical workstations over the next 2-3 years. Their specifications are:

- 30 MIPS processor
- 32 MB main memory
- 1 GByte hard disk with 17Mb/s (~2MB/s) write bandwidth
- 10Mb/s (~1MB/s) bandwidth interconnection (e.g. Ethernet)

4.2.4 Mirrored System Implementation

4.2.4.1 Synchronization Algorithms

During the implementation of the single computer algorithms it was realized that to ensure the same execution by two different processors (or the same processor during a recovery) as viewed from outside the node, it is sufficient to ensure that corresponding applications on two nodes have the same results for all system calls that they perform. For the majority of system calls executing on two nodes this will happen automatically. For example, an application executing on node A first makes an `open` system call for a file. The corresponding application on node B will make the same call and if the logical disk layouts of both nodes are identical both calls will have the same result.

There are two classes of calls which require special attention. The first type of calls are *interprocess* system calls. The majority of these calls are interprocess communication (IPC) related, but include other types such as signals, file system calls, and status related calls. When a process performs a call which is affected by, or affects another processes' execution the results depend not only on the execution of the process making the call, but also on the time the call was made with respect to the target(s) processes. In Figure 46 for example, if a kill signal is sent at time A process 1 will never make the call at time B. If the signal is sent at time C the call at B will have been made. The call, can in turn, affect the execution of other processes and as an end result affect the behavior of the node as viewed from outside the node. Similarly, for IPC calls, the results will depend on the order of the call with respect to other calls which can affect their outcomes.

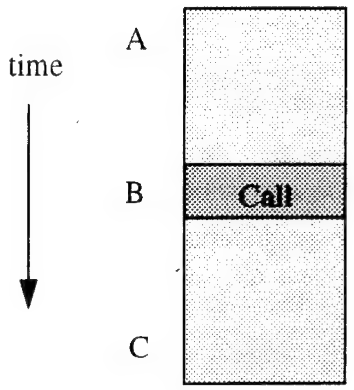


Figure 46. If process is killed at time A, call will not take place

If it were possible to synchronize the schedulers on the nodes via common interrupt and clocking signals this class would be automatically synchronized. However, that approach requires complex, platform specific hardware [4][16][33]. The SBMS approach is to synchronize at the event level in software. By introducing a minimal number of *execution synchronization points* (ESPs) into the operating system it can be guaranteed that interprocess calls will have then same results across an SBMS system.

The synchronization mechanism works on the principle that the process identifier of a process running on one node will match the process identifier of its corresponding process(es) on the other node(s). This will hold true as long as the processors remain synchronized because the process creation calls will always occur in the same order on all machines giving corresponding processes the same identifiers. Each node maintains a *pending synchronization list* (PSL) which is merely a list of process identifiers. When the operating system arrives at an ESP during execution of a call requested by a process it checks the PSL for the requester's identifier (note: from here on a process executing in the operating system on behalf of a user level process and the user level process will be considered the same process with the user level identifier serving as its process identifier). If it does not find the id in the PSL it sends the identifier to the other node(s) and then sleeps. Other processes on the node which are runnable can continue to execute and may in fact perform synchronization themselves. This first scenario arises whenever a process is not the last of its corresponding processes on the nodes to arrive at an ESP. The last process in a group of corresponding processes to arrive at an ESP will find its identifier in the PSL. It will then send the identifier to the other nodes to wake the sleeping process(es) up and continue without sleeping.

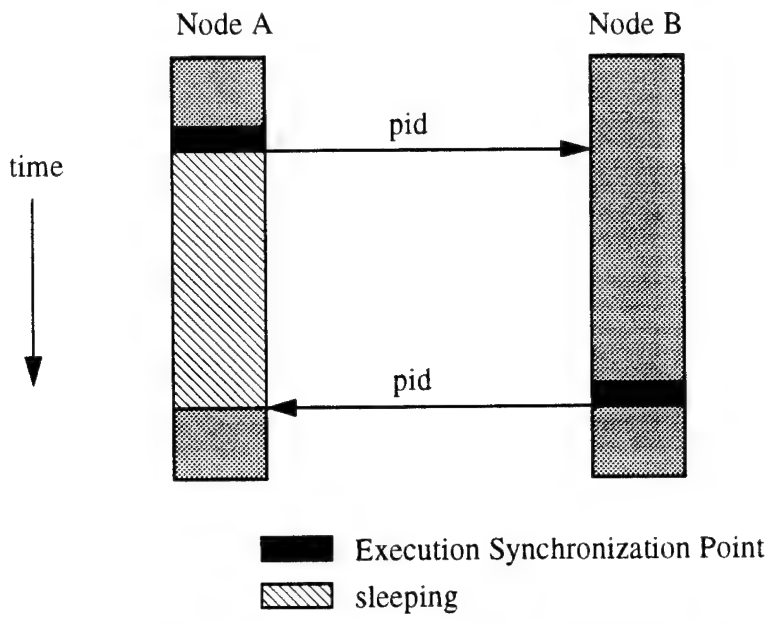


Figure 47. Example of execution synchronization point

Figure 47 shows an example. On the left side a process is executing on Node A. On the right the corresponding process is executing on Node B. The process on Node A arrives at an ESP first. Not finding its process id in the PSL it sends it to Node B and then sleeps. Node B receives the identifier and not finding the owning process asleep will store it into the PSL. Sometime later Node B arrives at the synchronization point. The process will find its identifier in the PSL so it sends it to Node A and continues. When Node A receives the identifier it will wake up the process owning it.

The second class of special system calls are *temporal* calls. These calls include ones which get the time of day and statistics on processor utilization of a process. Calls like this are dependent on hardware such as the system clock and the scheduling interrupt. Without very low level hardware synchronization these types of calls will return different results on different executions of the same application. To make these calls identical on different machines, a second type of synchronization point is introduced called a *data synchronization point* (DSP)

Each node maintains a *pending data list* (PDL). When a node arrives at a DSP it checks the PDL. If it finds its process identifier in the PDL it extracts any return data associated with the identifier and uses that data as the result of the system call. Essentially, the use of DSPs cause any temporal system call results to reflect the result of the first call.

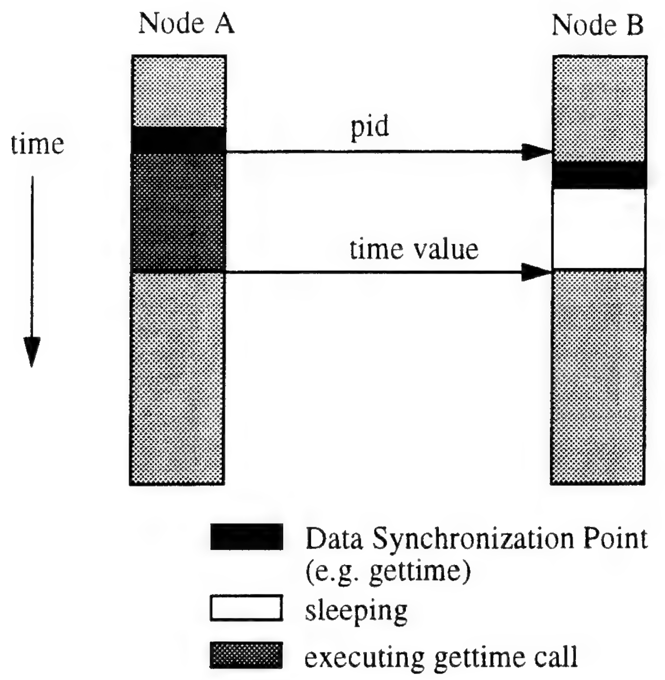


Figure 48. Example of data synchronization point

In Figure 48 there is again a two node SBMS system. Node A makes a *gettime* call. When it arrives at the DSP in the call it checks the node's PDL. Not finding its process identifier on the list it will send the process identifier to Node B and execute the call. Upon finishing the call it sends the time value it received to Node B which stores it in its PDL with the process identifier. When Node B executes the *gettime* call it will find its identifier in the PDL, but must wait for data from Node A. Once the data arrives it uses it as the time value for the call and then it will clear the data and identifier off the PDL and skip the actual call. To prevent time from potentially appearing to flow backward the node also sets the current time it uses to the system call result. It is possible for two nodes to essentially simultaneously reach a data synchronization point and for messages to cross each other in transit. To handle this case, processors are assigned a seniority number so that all nodes defer to the senior-most node as having been the node to reach the point first.

Synchronization of access to shared memory, which is supported by relatively new operating systems such as Solaris (Sun Computer), UNIX System V [3], and Windows NT [10], necessitates a different approach from system call synchronization. Shared memory access is not visible to the operating system. This, of course, is the reason it provides an efficient communications medium. In order to keep accesses on two machines synchronized it is necessary to use the operating system to intervene. All shared memory pages must be read and write protected for each process so that a fault occurs when a process attempts to access a shared page. The operating system is then aware of the attempted access and can synchronize with the other nodes in the system. Once a process has been granted access to the page, its access must be made atomic. This synchronization degrades the performance of intense shared memory applications, but it offers binary compatibility with no special hardware.

4.2.4.2 Synchronization Performance

In the single computer algorithms points similar to ESPs were placed in the UNIX operating system so that a concurrent application's behavior during rollback recovery would be identical to that of its initial execution. Experiments running applications which perform intense interprocess activity such as X and its client programs have shown peaks of about 10,000 of these points crossed in a one minute period with 300 temporal calls executed in the same period. Each ESP in SBMS requires a process identifier to be transmitted to other nodes. A process identifier in UNIX is two bytes in size. Temporal calls recorded in experiments require an additional four data bytes and to be conservative, a rate of 1000 temporal calls per minute will be used in calculations. Using the above rates a two processor system would require:

$$2 \times 10000\frac{\text{ESP}}{\text{minute}} \times 2\frac{\text{bytes}}{\text{ESP}} + 1000\frac{\text{DSP}}{\text{minute}} \times 6\frac{\text{bytes}}{\text{DSP}} = 46000\frac{\text{bytes}}{\text{minute}} \times \frac{1\,\text{minute}}{60\,\text{seconds}} = 767\frac{\text{bytes}}{\text{second}}$$

At 770 bytes/second the Ethernet bandwidth of 1MB/s is far less than 1% utilized. For maximum performance the ethernet connection requires low level drivers which implement a very basic protocol specialized for SBMS communication needs. If each synchronization point results in, as an upper limit, 1000 instructions to service, the additional load on a processor would be 20 million instructions per minute (the node would cross about 10000 points and have to service 10000 messages from its partner). The 1000 instruction count includes not only the actual number of instructions executed, but any overhead caused by blocking due to lack of tight coupling of the nodes. Given a 30 MIPS machine the degradation is about 1% for each node.

A three processor implementation would require:

$$3 \times 10000 \frac{\text{ESP}}{\text{minute}} \times 2 \frac{\text{bytes}}{\text{ESP}} + 1000 \frac{\text{DSP}}{\text{minute}} \times 6 \frac{\text{bytes}}{\text{DSP}} = 66000 \frac{\text{bytes}}{\text{minute}} \times \frac{1\,\text{minute}}{60\,\text{seconds}} = 1100 \frac{\text{bytes}}{\text{second}}$$

This is still far less than 1% of Ethernet bandwidth. The performance degradation of each node would be doubled to less than 2%.

4.2.4.3 Fault Detection Algorithms

In SBMS there are three types of fault detection which can take place:

- native processor and operating system fault detection
- fault detection through synchronization
- enhanced synchronization detection and inter-node data comparison

Coverage and effectiveness of native processor and operating system detection capabilities will not be addressed here, but it will be assumed that those mechanisms detect the vast majority of faults. A system fault detected by one of these mechanisms would cause notification of other nodes and a shutdown to occur.

The last two bullets are optional ways to increase fault detection coverage of nodes at varying costs. The basic synchronization algorithm presented earlier can be augmented to include a time-out mechanism. An upperbound would be placed on differences in how long it takes a node to reach a synchronization point after another node has already reached it. When a node is the first to reach an ESP or DSP it sleeps as before. If a time-out interval expires before the process is awoken by another node, the node would assume that the other node is no longer functioning or that its behavior is different from its own (it has become unsynchronized). In either case a fault has been detected. In a two processor configuration the fault is ambiguous and if the other node is still running it will also detect a difference in execution of the two nodes. In this situation not enough is known to determine why the behavior diverged. There are several things which can be done at this point:

- Arbitrarily shut down one processor for reintegration
- invoke increasingly more comprehensive test suites to self-test each node until the fault is detected
- let the system user decide which machine will be declared faulty

In the three node configuration, the faulty node will be the node that doesn't reach the synchronization points that the others do.

The synchronization can be augmented further with synchronization identifiers. In the above algorithms there is no check to make sure that when a process arrives at a synchronization point that its corresponding process on other nodes has arrived at the *same* synchronization point - only that it has arrived at *some* synchronization point. Of course, in a correct implementation the only time that this will not be true is in the presence of a fault. To make sure that all nodes have arrived at the same synchronization point each point is given a unique identifier which is transmitted with the process id. If a process arrives at a different synchronization point than its corresponding processes then a fault has been detected. This extra information will be one byte in size (there should be less than 256 synchronization points - the single computer implementation of Chapter 3 has about 15) increasing bandwidth usage by 1.6KB/s.

Comparison of system call results across different nodes allows for a further degree of fault detection. This fault detection mechanism would use fault detection points (FDP) at system call exits that would act like DSP except that data comparison would occur. Different levels of this mechanism could be invoked depending on system requirements. The first level would do a checksum comparison of selected outputs calls such as a write to the console or network. Higher levels would compare entire output streams and other system call results as well.

4.2.4.4 Integration Algorithms

Once a fault has been detected by the SBMS system, the faulty node must be replaced. In a two processor system the time between the fault and the point at which the new node is fully integrated is a window of vulnerability where the fault-free node is a single point of failure. A main goal of integration is to close this window as much as possible. As mentioned in the first section other goals are to have the replacement be an identical copy of the other nodes once integration is complete and to minimally disrupt the availability of the fault-free nodes.

There are three system components a new node must copy from a fault-free node:

- CPU state (processor flags, program counter, etc.)

- Main memory contents
- Secondary storage contents

The components are all brought up to the state of the functioning machines concurrently so they will be discussed with respect to the rolls they play in integration, rather than separately.

Integration of a new node would begin first with an initialization. The new machine is booted with the SBMS floppies. This process would load the memory resident parts of the operating system, and begin initialization of the hard disk. A considerable amount of space on a typical system is taken up by system files and executables. The SBMS disks would contain copies of the disk's initial system setup and also be brought up to date by the user to reflect new programs and directories that had been installed on the system. Thus, the initialization of a replacement node can be done before a running node has failed, cutting down on integration latency. In addition, this hard disk initialization will in many cases account for a sizeable percentage of the used portion of the disk on the running nodes.

Because there will be a portion of the disk that has not been made to reflect the state of those of the functioning nodes, all disk blocks that are unused in the initialization setup are marked specially as *copy from buddy* (CFB) on the integrating node and as *copy to buddy* (CTB) on the buddy node. All disk blocks belonging to directories which can be modified by running applications are also marked this way. This will be discussed in detail later.

After system integration has begun, as directed by the user, the new node and the buddy node will both move to integration code which has been mapped and wired (meaning that it cannot be moved to secondary storage) to the same physical memory location. The buddy node will then transfer over the connection all pages which are marked as wired. Note that these wired pages will also contain the operating system stack. The buddy must also mark all non-transferred memory pages as *wait for integration* (WFI) which will be discussed next.

The integrating node and the buddy node will then have the same state except that the majority of the integrating node's physical memory, and non-updated disk blocks have to be obtained when needed from the buddy. Figure 49 shows the state of an integrating node and buddy node upon completion of the initial integration procedure.

Once this initial state has been reached, both nodes continue with a *return-from-subroutine* instruction which will pop a context off of the stack and continue the operation that was in progress before integration started. Following initial integration the integrating node may try to access memory pages or disk blocks which are marked CFB. It will jump to the integrating code which requests these pages or blocks from the buddy. It is important to note that the buddy node

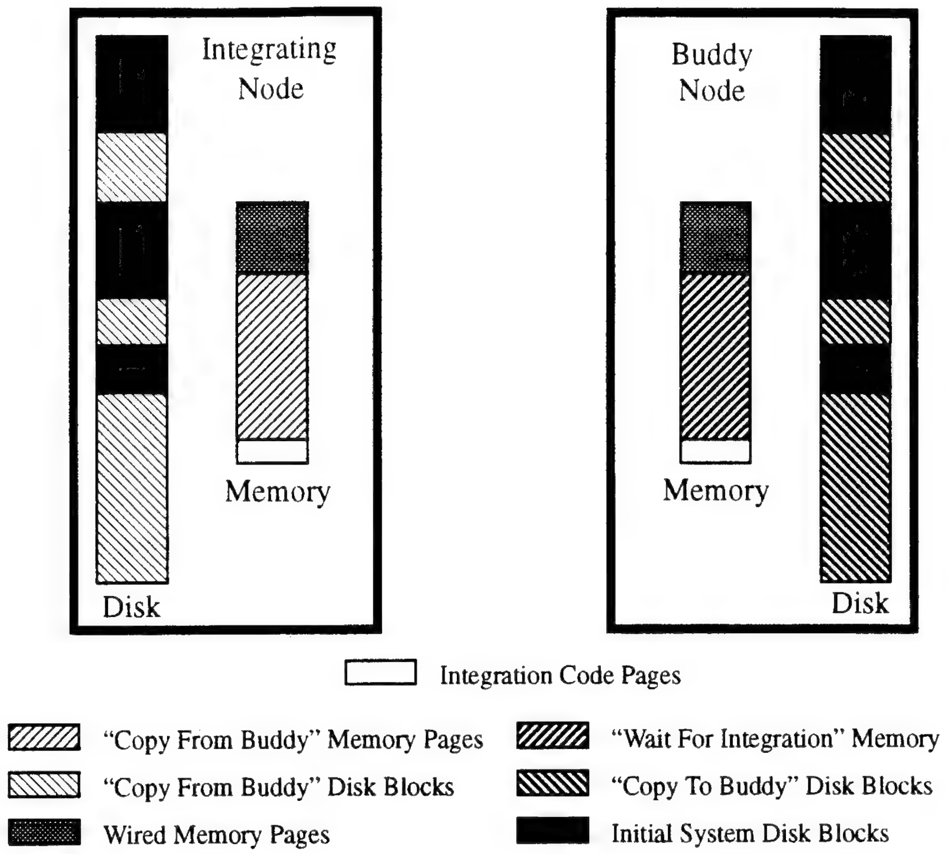

Integrating Node
Memory
Disk
Buddy Node
Memory
Disk
Integration Code Pages
"Copy From Buddy" Memory Pages
"Wait For Integration" Memory
"Copy From Buddy" Disk Blocks
"Copy To Buddy" Disk Blocks
Wired Memory Pages
Initial System Disk Blocks

Figure 49. Integrating node and buddy node after initial integration

cannot modify any of its WFI pages before they have been transferred to the integrating node. This prevents the integrating node from receiving dirty pages. Therefore, when the buddy tries to modify a WFI marked page it will page fault into the integrating code where it will pause until it has transferred the page to the integrating node. Any pages or disk blocks which have been transferred are cleared of their special designation and treated as in normal operation.

This algorithm will *eventually* lead to a completion of integration where the states of the buddy and integrating nodes are identical. However, because it is necessary to reduce integration latency as much as possible, an auxiliary transfer method must be in effect. As a low priority task the integrating code on the buddy will start at one end of memory and proceed to transfer any memory pages marked WFI to the integrating node. Once this is completed, it will start at one end of the disk and proceed to transfer all disk blocks marked CTB to the buddy. This process guarantees that within a short period of time the two nodes will converge.

Transfer of data from one machine to another will result in moderately degraded performance until integration is completed. For this reason all time-outs used for fault detection must either be relaxed or turned off for this period of time.

A note should be made here about copying disk images. In the past, maps of bad disk blocks were maintained by the operating system. If this was the case now, the disk integration routines would not work if two disks had different bad block layouts (which is likely). Modern disk drives are logical block addressed and the disk driver controller maps the logical block to a functional physical block. This means that the disk is logically copied and the logical block numbers are identical for the original and the copy.

4.2.4.5 Integration Performance

A large part of the integration actually takes place before the new machine is even connected to the functioning SBMS system. This involves booting off the SBMS system disks which performs hard disk initialization, prepares the node for integration and signals a buddy that it desires to become part of the existing SBMS system.

Once integration has started all wired memory pages must be transferred from the buddy to the integrating node. During this transfer period, the SBMS system appears dead. Even in a three node system, the node not participating in the integration remains paused during the initial integration. Pages that are wired include operating system code, data and stack. UNIX systems studied have between 2 and 4 MBs of wired memory. The transfer of this memory from the buddy is constrained to 1MB/s by the Ethernet resulting in a one second pause for every 1 MB of memory transferred. The rest of the memory is copied on demand or by the background copying mechanism which would use run-length data compression to increase the perceived bandwidth. An upper lower bound on the total time to copy memory is obtained by assuming that the copy is the only load on the system. In this case between 28 and 30 seconds is necessary. The actual time required would be determined by the actual system load, but a time limit would be placed on the duration causing a rise in the priority of the integration code for every fixed interval (e.g. five seconds) over thirty seconds. A reasonable cap would be one minute.

The disk integration would be handled in an identical manner, and would start after memory integration was completed. Current UNIX systems in use at CMU use about 150 MBs of disk storage for system files and executables which include the X window system. This will be the number used in calculations here. Because the disk is 1GB in size and initialization formats about 150MB, roughly 850MB must be copied from the buddy either through demand-request, or by the background integration code. Again, data compression will, in many cases, dramatically reduce the integration time. To obtain an upper lower bound on the time required to complete disk integration, it is assumed that the integration code is the only active task on the node for the duration of integration and that there is no data compression. If this is true then the transfer rate is again limited by the Ethernet bandwidth to 1MB/s as in the memory transfer case.

$$850\text{MB} \times \frac{1\text{second}}{1\text{MB}} \times \frac{1\text{minute}}{60\text{seconds}} = 14.2\text{minutes}$$

Thus, at a full rate of transfer, approximately fifteen minutes is required. The actual time required will vary depending on the system load. Of course, it is felt that for every fixed time interval that expires above fifteen minutes (e.g. every five minutes) the integrating code should be given higher and higher priorities to keep integration time to less than some cutoff such as thirty minutes.

In summary, system integration begins with a two to four second pause in service followed by a fifteen to thirty minute reduced performance period.

4.2.5 Comparison to Existing Fault Tolerant Systems

So as not to compare apples to oranges a comparison will be done of two processor systems to a dual SBMS (2SBMS) system and three processor systems will be compared to a triple SBMS (3SBMS) system. In the tables presented, items not included are either roughly equivalent on the compared systems or there is not enough information available to make a comparison.

4.2.5.1 Dual Mirrored Systems and Tolerance

A new entry into fault tolerance is Tolerance computer with their recent introduction of their two processor system. The Tolerance machine shares many characteristics with SBMS such as off-the-shelf nodes, but SBMS has several advantages. Table 9 gives a comparison of some of the key characteristics. SBMS is superior to Tolerance in performance. The performance figures quoted by Tolerance representatives and the requirement of a 250Mb/s connector indicate that SBMS synchronization is several orders of magnitude more efficient than that used in Tolerance. In addition, Tolerance has no provision for shared memory, has a fifteen second outage during integration versus two to four in SBMS, and it is unclear as to whether disk integration is even supported by Tolerance.

	Tolerance	2SBMS
off-the-shelf nodes	yes	yes
off-the-shelf interconnect	no	yes
shared memory support	no	yes
enhanced fault detection	limited	multiple levels
interconnect bandwidth	250Mb/s	10Mb/s
performance degradation	30-40%	1%
integration pause	10-15 seconds	2-4 seconds
new disk integration	?	yes

Table 9. Comparison of Tolerance and dual mirrored system

4.2.5.2 Triple Mirrored Systems and Tandem S2

The Tandem Integrity S2 [16], [33] is a UNIX fault tolerant machine sold by Tandem computer. It consists of three tightly coupled MIPS R2000 processors which are connected to two voters that are used to access peripherals. Not all details of the system are available. One interesting point is that when a processor fails, there is a 1-2 second pause in operation for integration of a new node. No literature discusses integration of a new disk.

	Tandem S2	3SBMS
off-the-shelf-nodes	no	yes
off-the-shelf interconnect	no	yes
shared memory support	yes	yes
performance degradation	?	2%
integration pause	1-2 seconds	2-4 seconds
new disk integration	?	yes
enhanced fault detection	very high levels	multiple levels

Table 10. Comparison of Tandem S2 and triple mirrored system

Tandem's main advantage over SBMS, as shown in Table 10, is fault detection and low level recovery. SBMS relies on native operating system and hardware fault detection whereas Tandem's machine has high levels of additional hardware fault detection and recovery. SBMS, however, can offer higher levels of fault detection at an increased performance penalty. Given that Tandem requires special purpose hardware whereas SBMS uses off-the-shelf chips, high-end SBMS systems can use cutting edge processors soon after they are made available. Tandem's chip technology is likely to lag several years behind. Another SBMS advantage is obviously the cost. At 1/5 (estimated) the initial investment, and with minimal replacement costs and maintenance fees, SBMS offers much lower price/fault tolerance.

4.2.6 SBMS and the Sentry Mechanism

None of the algorithms presented in this section or in Chapter 3, have any dependence on the sentry mechanism. However, the advantages of integrating the algorithms into the sentry mechanism framework are many, and covered in Chapter 2. The algorithms relating to SBMS are very similar to the single computer algorithms developed and implemented with the sentry mechanism. Instead of writing a global sequence number to a journal, SBMS has execution synchronization with its partners. The principle of operation is identical and the implementation relies on the sentry mechanism's operating system entry and exit control points as well as its bypass facility (as in the case of data synchronization points). This serves as another example of the sentry mechanism's perfect fit with the requirements of a complex, practical, and desirable fault management policy.

4.3 Distributed Systems

4.3.1 Introduction

Distributed systems are computing environments that consist of multiple processing nodes and an application that has cooperating components executing across these nodes. Figure 45 shows how fault management is introduced in such systems. The purpose of the investigation of these systems was to see how the sentry mechanism could be integrated with existing recovery algorithms. Distributed recovery algorithms are based on checkpointing or a combination of journaling and checkpoint. In algorithms where journaling is used the contents of messages are saved to stable storage so that if a node fails, its state can be restored by replaying the saved messages.

In the course of the investigation a new type of recovery algorithm was developed which has been called *hybrid checkpointing*. This new type of algorithm is based on the journaling of message order, rather than message content. To recover a failed node, all nodes that have communicated with the failed node are rolled back to their previous checkpoint and their execution is recreated by using the message ordering information. This technique is a direct extension of the sequencing algorithm of Chapter 3 to distributed systems. By saving message order instead of content, their is a potential savings in both stable storage requirements as well as journaling overhead. In order to make a recovery based on message order possible, a new form of checkpointing has been devel-

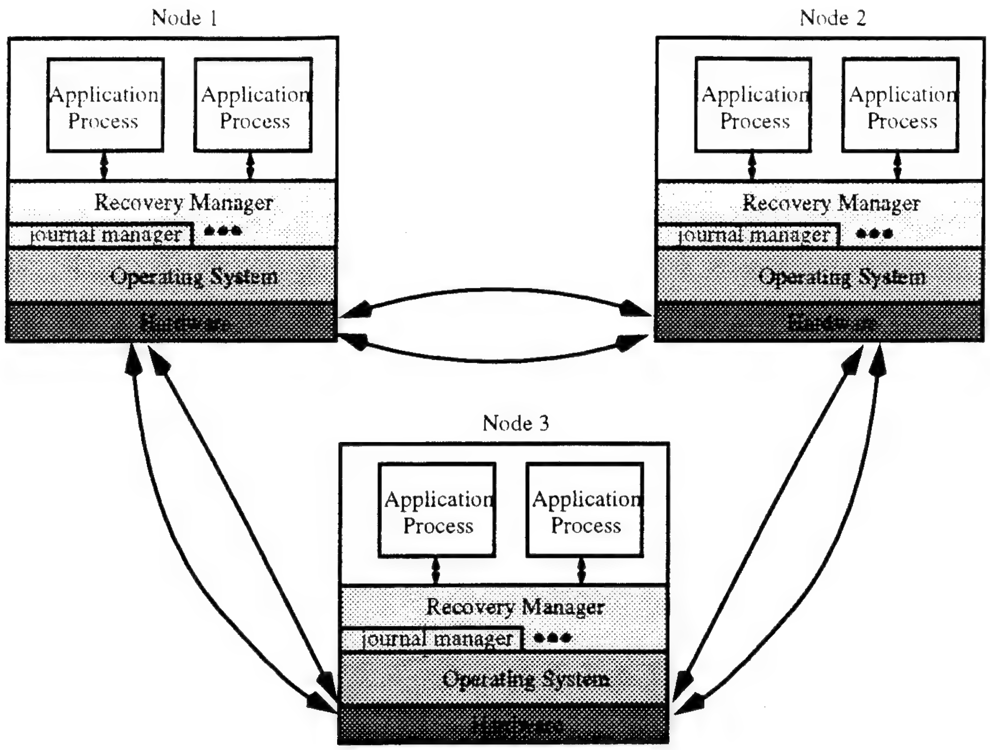


Figure 50. Architecture of three node distributed system

oped. The hybrid checkpointing algorithm provides nodes with the performance advantages of independent uncoordinated checkpointing while producing a consistent checkpoint. This section first presents the new algorithm and its variants, compares it to existing algorithms and then briefly discusses integration of distributed algorithms with the sentry mechanism. The tone of the section is slightly different than that of the rest of the thesis because the description of the algorithm has been formal in preparation for submission to a journal.

4.3.2 The Hybrid Checkpointing Algorithms

Different distributed recovery algorithms can be characterized by their fault-free overhead, the overhead of committing output [11] to the outside world and by the number of nodes that are required to undergo a rollback after one node has failed.

Fault-free overhead and the cost of committing outputs is considered here to be the areas where efficiency is most important because failures are relatively rare in modern systems. In addition, since tightly couple applications will have to be suspended until a failed node recovers, forcing multiple nodes to recover simultaneously will not greatly impact recovery latency. Existing algorithms make trade-offs between the three areas presented, typically doing well in one area, while suffering in another. These algorithms can be divided into three categories: pessimistic logging, optimistic logging and consistent checkpointing. Each category is briefly described here with respect to their characteristics. Throughout this section the term *recovery unit* (*RU*) [36] is used to refer to the smallest unit of computation that can be recovered. Typically, systems are based on the recovery of nodes or processors, but individual processes running on the same node can be treated as separate *RU*s with respect to a distributed recovery algorithm.

Pessimistic logging algorithms [7], [37] save message contents synchronously to stable storage. There are a few variations on this theme, but generally a node saves all incoming messages. Nodes are allowed to take checkpoints without coordinating with other nodes and if a node fails only that node is required to rollback to its previous checkpoint. After recovering the checkpoint, the node plays itself the saved messages and eventually recreates the state it had at the time of the failure. These types of algorithms are ideal in environments where writing to stable storage is inexpensive or communication is infrequent with respect to the overhead of writing to stable storage.

In optimistic logging algorithms [17], [18], [20], [34], [36], [37] message content is journaled asynchronously. Due to the asynchronous nature of the journaling, a node failure can result messages that were never stored in stable storage. These algorithms typically are only concerned with being able to recover the state that exists for particular events, most notable, the sending of messages to the outside world. The internal state of the system is considered of secondary importance, but any state that caused an external output to be produced must be recoverable. Typically, an *output commit* algorithm is executed that makes the state that lead to an output stable. Message dependency information is augmented to communication so that the commit algorithm can direct nodes that the output node's state depends on to flush their journals to stable storage. In these algorithms, a failure can result in non-faulty nodes being forced to rollback.

Consistent checkpointing algorithms [8], [21], [24], [37] are based entirely on checkpointing. When a state must be made recoverable, all nodes take a consistent checkpoint. While no other form of overhead is present in these algorithms, taking a consistent checkpoint, described in detail later, is an expensive operation that requires all computing to be suspended for the duration of the checkpoint.

The new algorithms are both based on the same basic principles. The first algorithm, called the *pessimist-hybrid* algorithm, is tuned for systems where system output is frequent. As in pessimistic logging algorithms, there is no overhead for committing external outputs and there is a degree of synchrony in journaling. It is better than pessimistic logging, however, because message *order* is journaled, rather than message *content*. Also, synchronous journaling to stable storage can be avoided given certain conditions. These factors can result in savings of several orders of magnitude in the amount of stable storage required and can often result in improved fault-free performance.

The second algorithm, called the *optimistic-hybrid* algorithm, is tuned for systems for which internal communication dominates the rate of external output. This algorithm shares characteristics with optimistic logging in that journaling is asynchronous, reducing fault-free overhead, and a multi-host protocol is executed when external output is made. Optimistic logging algorithms journal message content while the optimistic hybrid algorithm again only journals information on message ordering, resulting in reduced stable storage requirements and increased performance. The output commit protocol requires only loose synchronization of nodes and is based on the well known two-phase commit algorithm.

Both algorithms share a new checkpointing scheme that has characteristics in common with both asynchronous and synchronous checkpointing. When a *RU* checkpoints without coordination with other *RU*s, as in pessimistic logging, it is called an asynchronous checkpoint. Synchronous checkpointing is performed by consistent checkpoint algorithms and results in a halt of all processing while all *RU*s take a checkpoint. The checkpointing algorithm introduced here is called *hybrid* checkpointing because it offers the performance advantages of asynchronous checkpointing, but requires multi-host coordination like synchronous checkpointing. This section presents each algorithm and proves it correct.

4.3.3 Assumptions

It is assumed that the system consists of *n RU*s that form an application or a *system*. Each *RU* has a unique identifier and there is a seniority function, such as identifier value comparison, which can be applied to the identifiers. *RU*s in actuality do not need to reside on different processors, but their communications are restricted to message passing. An *RU* can consist of one or more threads of execution whose basic behavior can be non-deterministic. Like in the algorithms of Chapter 3, any non-deterministic events have information saved to the journal so that they can be recreated. In the course of describing the algorithms, *RU*s are often referred to as taking particular actions to write to the journal or recover their state. In reality, sentry policies are performing these actions on behalf of the RUs.

It is also assumed that *RU*s can communicate with any other *RU*. Unlike the majority of existing rollback-recovery algorithms, however, the communications channels are not assumed to be reliable. Reliable communications protocols are part of the algorithms themselves. Conceptually, one can think of each *RU* as having a communications handler, CH, that receives messages from the network, queues them, acknowledges them and delivers them to the application in the correct order. Sequence numbering and an algorithm such as the sliding window protocol is used by the *CH* to transparently provide the application FIFO communications channels. Like most rollback-recovery algorithms partitions of the physical system are not handled. Importantly, the algorithms can recover even in the presence of multiple simultaneous failures.

Each *RU* must have access to a stable storage device with enough space to hold two complete checkpoints plus journaled information. When a node fails, it is assumed to do so in a fail-stop manner and all healthy *RU*s are informed of the failure. The faulty node begins a recovery procedure to restore the state of the failed *RU*s that were running on the node and if the node is permanently failed, another node must be given access to the failed node's stable storage in order to recover the failed *RU*s.

4.3.4 The Pessimistic-Hybrid Algorithm

4.3.4.1 Overview

In the pessimistic-hybrid algorithm, *RU*s synchronously record information about the messages they receive in stable storage. Periodically, a *RU*, or *RU*s, will determine that a global checkpoint should be taken and at that time initiate the execution of the hybrid checkpoint algorithm. Once the checkpoint algorithm is complete, the system has a consistent point to rollback to upon a failure and the previous checkpoint and journal are deleted. When a failure occurs, all *RU*s must rollback to the most recent committed checkpoint and begin re-execution. *RU*s replay their execution by ordering incoming messages the same way they did in the original run.

This section presents the components of the algorithm related to reliable communications, fault-free execution, external output, checkpointing, and recovery. The concluding part of the section contains proofs of the algorithm's correctness.

4.3.4.2 Handling Communications Errors

Because the communications medium is not assumed to be reliable, the algorithm must make allowances for errors. This is done with the standard acknowledgment protocols. All messages received by a *RU* are acknowledged when they are received. When a message is sent by a *RU*, the *RU*'s *CH* stores the message content in volatile memory until it receives the message's acknowledgment. If it does not receive the acknowledgment within a specified time-out period, the message is resent. The CH also uses a sequencing algorithm such as the sliding windows protocol to enforce FIFO communications. All messages, including messages that are part of the checkpointing protocols, are passed through the CH. The internal messages must use different sequence numbers, however, so that they are transparent with respect to the sequencing of application messages.

4.3.4.3 Fault-Free Execution

Each RU_i maintains a *checkpoint interval number*, CI_i, a *send message sequence vector*, MS_i, with an entry for each *RU* in the system, and a *receive message sequence vector*, MR_i. When a *RU* begins taking a new checkpoint it increments the checkpoint interval, and when it sends a message to RU_j, it increments the message sequence number it keeps for the target *RU*, $MS_i[j]$. The checkpoint interval and the send message sequence number $MS_i[j]$, are both appended to messages sent to other *RU*s. In addition, if the communications system does not already allow a receiver to identify the sender, the *RU* identifier must also added. This additional information overhead constant, or O(*1*) in size.

When RU_i, receives a message from RU_j $(i \neq j)$, it notes the logical time at which the message was received, the sender's identifier, and the message sequence number. The logical time is a value which indicates the logical point in the *RU*'s execution it received the message. For example, if messages are received synchronously, then the count of the number of system calls executed before the message was received can serve as the logical time reference. These pieces of information together form a journal record. The *RU* also saves the received sequence number in the sender's entry of the receive sequence number vector, $MR_i[j]$, and then initiates an asynchronous flush of the volatile data to stable storage with the restriction that the data is fully flushed before any messages are sent by the *RU*. This is where the algorithm gets the name pessimistic: journaling is done in a synchronous manner like the synchronous journaling performed by pessimistic recovery algorithms.

4.3.4.4 Output Commit

The pessimistic hybrid algorithm does not require any form of output commit when a message is sent to a receiver external to the system. Information on stable storage is sufficient to recreate the state of the *RU* at the time of the output. However, if it is not possible to resend messages to the outside world during recovery (e.g. sending output to a printer), then the *RU* must synchronously record in stable storage the fact that the output was made so that it is not remade during recovery.

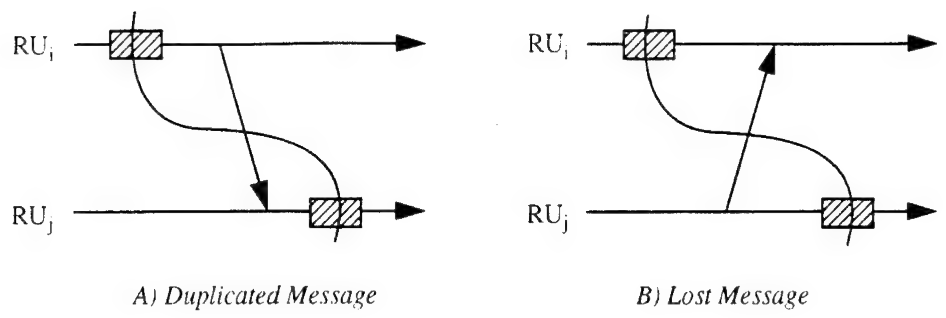
4.3.4.5 Hybrid Checkpointing

Periodically, one or more *RU*s will decide to initiate a checkpoint. The time between checkpoint sessions is called a *checkpoint interval*. The issues related to checkpoint interval are the same as presented in Chapter 3. Existing methods of determining the checkpoint interval include intervals based on elapsed time, limiting the size of the journal, and on external stimulus such as user direction. A *RU* considers itself the *checkpoint leader* CL_n, for checkpoint interval n, until its forced to give up this roll to another *RU* or it completes the checkpoint session. If more than one *RU* believes its the checkpoint leader, *RU*s defer in a deterministic manner to a single leader through a method called *leadership seniority*. One possible approach, and the method used here, is to defer leadership to the *RU* with the lowest *RU* identifier.

The leader *RU* begins a two phase commit protocol by sending *checkpoint initiate* messages to all the other *RU*s instructing them to take a checkpoint. When a *RU* receives this initiate message, it takes a checkpoint of its state by saving its send and receive message sequence vectors, checkpoint interval number, as well as necessary registers, virtual memory state, and data structures such as the memory map, process control block and communications connection structures. After the checkpoint begins, a new journal is begun and after the checkpoint is entirely in stable storage, the *RU* sends a message to the checkpoint leader indicating that the checkpoint is stable. This message is called the *checkpoint prepared* message. When the leader has collected a checkpoint prepared message from all other *RU*s, it sends a *checkpoint commit* back to all the *RU*s. A *RU* receiving a commit message can then mark the checkpoint as committed and delete the previous checkpoint and journal.

If a *RU* believes RU_j is the current checkpoint leader and it receives another checkpoint initiate message from RU_k, it applies the seniority function to determine if RU_k is senior to RU_j. If it is, it sends its checkpoint prepared message to the new leader, RU_k. If it has already sent a checkpoint prepared message to RU_j, it sends another one to RU_k. Otherwise it ignores the new request. A leader that receives a checkpoint initiate message from a senior *RU* will immediately stop its checkpoint protocol and act as a participant in the checkpoint with the senior *RU* as the leader.

By itself, this type of checkpoint can lead to an inconsistent checkpoint state [8]. Consistency will not be formally treated here as it has been dealt with extensively in literature. Consistency implies that the *RU*s must agree as to which messages have been sent and which have been received. The two types of inconsistent checkpoint are shown in Figure 51. The checkpoint in *A* leads to a message that RU_i has received, but that RU_j believes it hasn't sent. After the *RU*s recover, RU_j will send the message again to RU_i. The message in *B* is called lost because after *RU*s rollback to their checkpoints RU_j will believe that it has sent the message, but RU_i will not have received it.



▨ *Checkpoint*

Figure 51. Inconsistent checkpoints

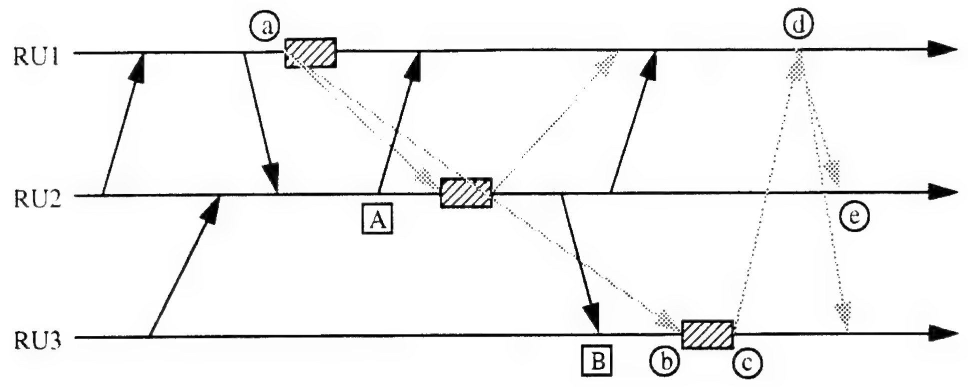
These two cases can be recognized during an execution by the receiving RU_i because in such a case the checkpoint interval number in the received message, CI_m, is different from CI_i. Each case is dealt with specially to make a potentially inconsistent checkpoint consistent. If $CI_i < CM_m$ (case A), the reliable communications protocol causes the message to be ignored during recovery because the restored receive message sequence vector will indicate that the message has already been received.

If $CI_i > CM_m$ (case B), there are two subcases that must be handled. In the first subcase, CH_i receives the message, stores the contents in stable storage with the journal, and then sends an acknowledgment to CH_j. If a failure occurs, RU_i will have the message replayed to itself by its CH during recovery. The second subcase is when, after a checkpoint has been committed, network delay results in messages that are lost because they have not been received at RU_i and therefore have not been saved to stable storage by RU_i. To insure that messages of this type are resent, all messages sent by a RU that have not been acknowledged at the time the RU takes a checkpoint are saved along with the checkpoint. If the checkpoint is committed and a rollback takes place these messages are resent and treated as unacknowledged.

This checkpointing protocol is called hybrid because it obtains a consistent checkpoint like synchronous consistent checkpoint algorithms do, but it obtains performance comparable with independent asynchronous checkpointing. Figure 51 shows an example execution of the hybrid checkpointing protocol.

4.3.4.6 Recovery

When a RU fails, the entire system begins a recovery. If a failure occurred during a checkpointing session, RUs may have two checkpoints in stable storage, C_p and C_{p+1}. C_p will always be marked



▨ *Checkpoint*

→ *Application Message*

⇢ *Checkpoint Protocol Message*

ⓐ *Checkpoint leader sends checkpoint initiate messages to all other RUs*

ⓑ *Participant receives message, increments checkpoint interval, and takes a checkpoint*

ⓒ *Checkpoint is stable so checkpoint prepared message is sent to leader*

ⓓ *Leader has received all prepared messages: checkpoint commit is sent*

ⓔ *RU receives a checkpoint commit and can delete the old checkpoint*

$\boxed{A}$ *Message has a lower checkpoint interval number than receiving RU so its contents must be stored in the receiver's new checkpoint so that the message can be replayed in recovery*

$\boxed{B}$ *Received message has a higher checkpoint interval than RU so information is stored in the RU's new checkpoint indicating that the message is ignored on recovery*

Figure 52. Example execution of hybrid checkpointing protocol

committed, but C_{p+1} may or may not be committed, and even if it is not marked committed, it may in fact be the case that the checkpoint is committed, but that the *RU* failed before it updated the checkpoint's status. To obtain the correct checkpoint to rollback to, all *RU*s send their most recent committed checkpoint's checkpoint interval number to a designated node such as the most senior *RU* or the node that failed (an alternative if there are relatively few *RU*s is to broadcast the number to the other *RU*s). If the senior node receives a message indicating that C_{p+1} is committed it can tell the other nodes to rollback to C_{p+1} because it means that it was committed. If no message is received indicating C_{p+1} was committed, the *RU*s must delete C_{p+1} if they have it in stable storage, and rollback to C_p.

Along with the rest of the *RU* state, a recovering *RU* restores the checkpoint interval number and the send and receive message sequence vectors that are stored in the checkpoint. It then begins execution. All outgoing messages are sent again (see the discussion of output messages in Section

4.3.4.4) and the *CH* for each *RU* reads the journal from stable storage and forces messages that are received to be delivered in the same order and at the same logical time as is recorded in the journal. For example, messages from different *RU*s may arrive at a receiving *RU* in a different order than recorded in the journal. The *CH*, however, delivers the messages to the application in the recorded order and at the correct logical time. Messages that were not recorded in the journal are buffered until recovery is complete and once a *RU* has been delivered all the messages recorded in the journal, it is delivered any buffered messages and journaling continues.

In some cases it may be desirable to only rollback the *RU*s upon which the failed *RU*'s recovery depends. An additional dependency vector can be kept and added to the journal records and then used to determine the *RU*s that must rollback [38]. This is considered an optimization that has been investigated in previous algorithms and is not be covered here.

4.3.4.7 Proof of the Pessimistic-Hybrid Algorithm

To prove that the algorithm is correct it must be proven that only one *RU* will ever commit a checkpoint, that the algorithm generates a consistent global checkpoint, that the local *RU* state that produced a message can be exactly recreated after a failure, and that the journal plus the checkpoint will result in a consistent state (i.e. that a recovering *RU* will never halt waiting for a message that will never arrive).

Lemma 4. *At most one RU will send a checkpoint commit message for a given checkpoint.*

Proof. By contradiction. Without loss of generality, assume that there are two *RU*s, RU_i and RU_j, with RU_i senior to RU_j. Assume that they both send checkpoint commit messages. This implies that each of them have received checkpoint prepared messages from all other *RU*s including each other. If RU_i has begun checkpoint session *p* when it receives the checkpoint prepare message for checkpoint *p* from RU_j, it will not send a checkpoint prepared to RU_j because it is senior and therefore considers itself the leader. If it hasn't started a checkpoint, it will take a checkpoint upon receiving the checkpoint prepare, and increment CI_i to *p*. When it starts a checkpoint, CI_i will be incremented to $p+1$, and therefore it will not lead the same checkpoint as the one RU_j lead. Hence, only one will send the checkpoint commit message for checkpoint *p*.■

Lemma 5. *The hybrid checkpointing protocol creates a consistent global checkpoint.*

Proof. Consistency implies that there will be no lost or duplicated messages when a rollback to the checkpoint takes place. In the case that there is no communication across from one checkpoint interval *p* to checkpoint interval $p+1$ the checkpoint will be consistent. There are two inter-interval cases to be dealt with. Consider two *RU*s, RU_i and RU_j, with a message, *m*, sent from RU_i to

RU_j. The acknowledge of the message receipt is ack_m. The checkpoint interval sent with the message is denoted as CI_m. Messages sent from RU_i to RU_j are assigned the sequence number $Seq_{i \rightarrow j}(m)$.

case 1: $CI_i < CI_m$. When the message is received by RU_i, $MR_i[j] \leftarrow Seq_{i \rightarrow j}(m)$. When RU_i takes checkpoint $p+1$, it saves MR_i. After a failure, RU_i restores MR_i and RU_j begins deterministically executing so that the message will be assigned the same sequence number, $Seq_{i \rightarrow j}(m)$. When it arrives at RU_i it will be discarded because $MR_i[j] \geq Seq_{i \rightarrow j}(m)$.

case 2: $CI_i > CI_m$. There are two subcases to consider:

subcase 1: when RU_j takes checkpoint $p+1$, it has not received ack_m. When RU_j checkpoints it saves the contents of m to stable storage. When it recovers it treats the message as an unacknowledged message and resends it. Note that the receiver always journals messages with $CI_i > CI_m$ so if RU_i had saved the message before it rolled back it will replay it to itself during recovery and using sequence numbers, ignore the resent message. If it did not save the message then it will fully recover and then receive and acknowledge the resent message.

subcase 2: when RU_j takes checkpoint $p+1$, it has received ack_m. The contents of m will not be saved by RU_j at the checkpoint, but the acknowledgment implies that RU_i has already saved the message to stable storage and therefore can replay it during recovery.

Since the hybrid algorithm results in no duplicated or lost messages, a consistent checkpoint is guaranteed. ■

Lemma 6. *The local RU state that resulted in a message being sent will be exactly reproduced after a rollback.*

Proof. A *RU*'s state is created deterministically with the exception of message receipt and non-deterministic events such as inter-thread synchronization. For a state to be recreated all messages that the state depends on must be received at the same logical time as in the fault-free execution. At the time a message, m_s, is sent by RU_i, it has completed saving information on stable storage that relates to all messages, m_r, that it has received and the outcomes of all non-deterministic events it has executed. During recovery the *RU* depends on other *RU*s to resend the messages. It buffers messages received before the journaled logical time and blocks waiting for messages that have not yet arrived when it reaches the recorded logical time guaranteeing that it will receive the messages at the same logical time. When it executes a non-deterministic event it forces the outcome to match the one recorded in the journal. Once the recovering RU_i has received all messages, m_r, and executed all non-deterministic events which lead to the state that produced m_s, m_s will be recreated. Induction is used on all *RU*s, RU_r, which sent the messages, m_r, implying that they will resend the messages, m_r, during recovery. Therefore, all *RU* states that resulted in a message being sent will be recreated after a rollback. ■

Theorem 3. *The combination of the pessimistic-hybrid journals and most recent committed checkpoint results in a global consistent state.*

Proof. Using Lemma 5 it is guaranteed that the rollback point is consistent. By applying Lemma 6 it is assured that the state in the journal will result in all messages being resent during recovery. This implies that no messages will be lost. The other consistency condition is that no messages will be received twice by a *RU*. This case is handled by the communications protocol which causes messages with duplicated sequence numbers to be ignored. Therefore, the state resulting from the checkpoint and the journal will be consistent. ■

4.3.5 The Optimistic-Hybrid Algorithm

4.3.5.1 Overview

The optimistic-hybrid algorithm is similar to the pessimistic-hybrid algorithm because it uses the same hybrid checkpointing and reliable communications protocol. What is different is that instead of synchronously journaling message order information to stable storage, the ordering information is asynchronously journaled and a commit protocol is used to commit new journal contents. The state of the system as seen from the outside can always be recreated if the journal commit protocol is applied before a message is sent to the outside world. This form of journaling raises the amount of information that must be added to messages from O(*1*) in size to O(*n*) in size. The algorithm is more efficient than the pessimistic-hybrid algorithm in systems where external output is infrequent with respect to communication among *RU*s because synchronizing with stable storage only occurs when output must be committed. This section discusses the behavior of the algorithm during fault-free execution, output commit, and checkpointing. It concludes with proofs of the algorithm's correctness.

4.3.5.2 Handling Communications Errors

The communications handler functions are identical for both algorithms (see Section 4.3.4.2).

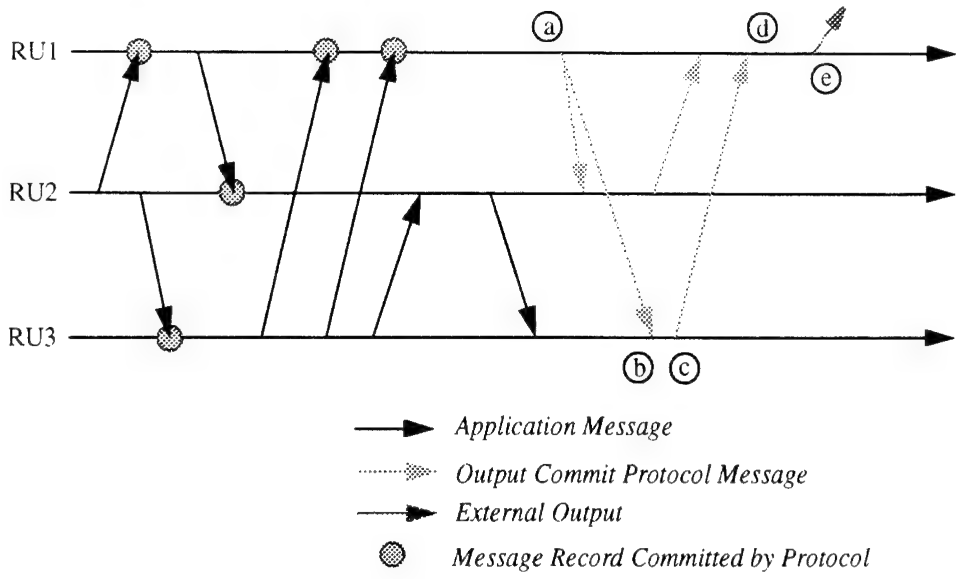
4.3.5.3 Fault-Free Execution

Like in the pessimistic-hybrid algorithm, each RU_i maintains a *checkpoint interval number*, CI_i, a *send message sequence vector*, MS_i, and a *receive message sequence vector*, MR_i. In addition, the *RU*s maintain a *global sequence number*, GS_i, and an *uncommitted dependency vector*, UD_i. $\mathrm{GS_i}$ is incremented each time RU_i sends a message. This differs from the values in MS_i which keep track of the number of messages sent to each *RU*. UD_i has entries which are global sequence numbers of other *RU*s that the current state depends upon and that haven't been committed to stable storage. It is initialized to all null values. *RU*s append their global sequence number and uncommitted dependency vector to all outgoing messages and when RU_i receives a message it updates UD_i by comparing each entry with the received vector, UD_m. If $UD_m[n] > UD_i[n]$, it copies $UD_m[i]$ to $UD_i[n]$. It stores the same message receive information that the pessimistic-hybrid algorithm records in volatile storage with the addition of the GS_i. This is where the algorithm gets the name optimistic: like optimistic logging algorithms, journaled information is saved in volatile memory and can be asynchronously copied to stable storage and like most optimistic logging algorithms, $O(n)$ information is added to messages.

4.3.5.4 Output Commit

When a *RU* is about to send a message to the outside world it first executes a multi-host commit protocol in order to guarantee that the state that produced the output can be recovered after a failure. The *RU* initiating the journal commit, RU_i, is called the *journal commit leader* (referred to as the *commit leader*). The commit leader begins a flush of any journal it has in volatile memory and then examines UD_i and sends a *journal commit* message to all *RU*s, RU_n, such that $UD_i[n]$ is not null. The journal commit message contains $UD_i[n]$ and indicates that RU_i's current state depends on RU_n's state that lead to the sending of the message with the global sequence number $UD_i[n]$. When a *RU* receives a journal commit message it makes sure that all journal records that were stamped with global sequence numbers less than or equal to the number sent in the journal commit are saved in stable storage. It then updates a value in the journal called the *committed sequence number*, CS_i, and sends a *journal committed message* back to the leader. CS_i is kept in stable storage and indicates the highest journal record that has been committed. When the leader has received back all the expected journal committed messages, it marks all UD_i entries null and can proceed with the output. Note that the *RU* will not process any more received messages until after the output has been made.

A *RU* can at any time asynchronously flush its volatile journal to stable storage but the flushed data is not committed until the commit protocol is executed. Increased performance can result when data is already in stable storage at the time a *RU* receives a journal commit so *RU*s should asynchronously flush volatile journal records to stable storage as they are created. Figure 51 shows an example execution of the output commit protocol.



RU1
RU2
RU3
a
d
e
b
c
Application Message
Output Commit Protocol Message
External Output
Message Record Committed by Protocol

ⓐ *Journal commit leader sends journal commit messages to RUs it depends on*

ⓑ *Participant receives message, flushes messages up to the sequence number indicated, and updates the committed sequence number in stable storage*

ⓒ *A journal committed message is sent back by participants*

ⓓ *Leader has received all committed messages; it can now send external output*

ⓔ *External message is sent*

Figure 53. Example execution of output commit protocol

4.3.5.5 Hybrid Checkpointing

The checkpointing protocol followed by the optimistic-hybrid algorithm is identical to that described in Section 4.3.4.5. At a checkpoint, *RU*s record their current global sequence number and their message sequence vectors (note that the uncommitted dependency vector is *not* saved). The uncommitted dependency vector is not reset at this time because the checkpoint may not succeed so dependent state information cannot be assumed to be stable when a *RU* decides to do output. Between the point that a *RU* takes a checkpoint and the time that it commits the checkpoint it will have two journals, one associated with the old checkpoint and a new one it begins after taking the new checkpoint. If it becomes an output commit protocol participant in this time window, it may have to commit parts of both journals. Once the checkpoint is committed, the old journal can be deleted.

4.3.5.6 Recovery

When a node fails all *RU*s rollback to their committed checkpoints, delete uncommitted checkpoints and re-execute as described in Section 4.3.4.6. If a failure takes place during the checkpointing protocol a *RU* may have two journals, one associated with the committed checkpoint and one with the uncommitted checkpoint. The *RU* first reads the committed sequence number from the most recent journal and uses the committed part of the journal(s) to guide its execution. Once all messages with committed records have been received by the *RU* as directed by the *CH*, the *RU*'s recovery is complete.

4.3.5.7 Proof of the Optimistic-Hybrid Algorithm

The same conditions proven in the pessimistic-hybrid proofs must be proven for the optimistic-hybrid algorithm. Since the checkpointing protocol is identical for both algorithms the proof of checkpoint consistency will not be repeated here. The two conditions that must be proven are that state which resulted in an external message can be recreated and that the combination of the journal and the checkpoint will result in a consistent global state.

Lemma 7. *The local RU state that resulted in an output message being sent will be exactly reproduced after a rollback.*

Proof. A *RU*'s state is created deterministically with the exception of message receipt and non-deterministic events such as inter-thread synchronization. For a state to be recreated messages must be received at the same logical time as in the fault-free execution. At the time an output message is sent the *RU* has completed saving all local information with respect to message receipt and non-deterministic event outcome on stable storage. If the *RU* state at the time of output depends on some state achieved at another *RU*, the uncommitted dependency vector will contain the global sequence number of the other *RU* upon which the output state is dependent. Transitive dependencies are kept track of using the uncommitted dependency vector update routine. Therefore, the *RU* will know the maximum global sequence number of the other *RU*s upon which it depends. The output commit protocol insures that output is not made until all of these states have been committed to stable storage. Therefore, the state that the output depends upon will be recreated.■

Theorem 4. *The combination of the optimistic-hybrid journal and the checkpoints result in a global consistent state.*

Proof. Using Lemma 5 it is guaranteed that the rollback point is consistent. By applying Lemma 6 it is assured that the state that lead to system output can be recreated. What must be guaranteed is that no *RU* will wait on a message which is will not be delivered in recovery. This, too, follows from Lemma 6. If a *RU* is waiting for a message to arrive during recovery it means that the journal record associated with the message was committed to stable storage. Since the only time a record is committed is in the output commit protocol, and since transitive dependencies of any record committed are also committed, the state that resulted in the message being sent will be recreated. The other consistency condition is that no messages will be received twice by a *RU*. This is guaranteed by the *RU*'s *CH* in the manner described in Lemma 3. Hence, there will be no duplicated messages. Therefore, the state resulting from the checkpoint and the journal will be consistent. ■

4.3.6 Comparison With Existing Algorithms

In this section a qualitative comparison of the pessimistic-hybrid and optimistic-hybrid algorithms with existing algorithms is made. This task is difficult because it is rare for different algorithms to be based on the same assumptions. For this reason actual algorithms are not used; instead, the categories of algorithms mentioned in the introduction to this section: pessimistic logging, optimistic logging and consistent checkpoints. Optimistic logging refers to receiver based logging as sender based logging is not *n*-fault tolerant. One exception to the categories is the Manetho [11] algorithm which is based on assumptions identical to the ones that the hybrid algorithms are based on. All the algorithms are compared on the basis of what kind of data must be journaled, how journaling is performed, the amount of data which must be appended to messages, the output commit protocol performed, the checkpointing protocol performed and the extent of rollback. Table 11 summarizes the comparison. The extent of rollback is considered to be a fea-

Algorithm Type	Data Journaled	Journal Method	Message Overhead	Output Commit	Checkpoint Method	Extent of Roll-back
Pessimist Logging	content	sync	none	none	async	failed
Optimistic Logging	content	async	O(n)	coord. sync	coord. sync	> failed
Consistent Ckpt	none	none	none	cnstnt. ckpt	sync	all
Manetho	content	async	no bound	none	async	failed
Pessimistic-Hybrid	order	semi-sync	O(1)	none	hybrid	all
Optimistic-Hybrid	order	async	O(n)	coord. sync	hybrid	all

Table 11. Qualitative comparison of rollback recovery algorithm types

ture of secondary importance because if all *RU*s are recovering simultaneously, the number of *RU*s recovering will have little impact on recovery latency. This measure will not be used in comparisons.

As mentioned before, the pessimistic-hybrid scheme is tuned for systems with a high degree of external output. The other recovery algorithms that are tuned for this are pessimistic logging and Manetho. As mentioned earlier, pessimistic logging algorithms require that message content be journaled. Manetho also requires message content to be journaled, and if the system communications rate is high Manetho must frequently write the journal to stable storage to prevent message overhead from growing out of control. In addition, Manetho must flush the journal to stable storage before external output is made. Pessimistic-hybrid requires a coordinated checkpoint, but the protocol provides performance similar to asynchronous checkpointing.

Algorithms which are tuned for systems with infrequent output are optimistic logging, consistent checkpointing, Manetho and optimistic-hybrid. Consistent checkpointing is rarely considered practical because consistent checkpointing is expensive and must be executed any time an external output is made. Optimistic logging and Manetho both require entire message content to be journaled, while optimistic-hybrid only journals message order information. Optimistic logging algorithms require a coordinated output commit and Manetho requires the sender's journal to be synchronously flushed to stable storage. The output commit protocol of optimistic hybrid has efficiency comparable with optimistic logging output commit protocols and checkpointing overhead is again comparable with asynchronous checkpointing.

In both cases, the major strengths of the hybrid algorithms are that the amount of data that needs to be journaled is potentially orders of magnitude less than that required in comparable algorithms, and that the hybrid checkpointing provides an efficient method for obtaining consistent global checkpoints.

4.3.7 Distributed Algorithms and the Sentry Mechanism

The algorithms discussed in this section were designed from their inception to be application-transparent. Since the common feature of all of them is communications journaling and replay, it is that facet that must be evaluated in terms of implementation as a sentry. If a system call that is communication related is viewed as an external input, than proof that information about the call can be easily journaled by a sentry is seen in the single computer implementation of external device journaling. Likewise, the recovery of this journaled data would be performed in a manner analogous to replaying external inputs from a journal as is done in the single computer recovery.

The fact that the actions required by distributed algorithms fits so neatly into the sentry mechanism indicates that the distributed system algorithms can be combined with the single computer algorithms resulting in a *RU* model that consists of concurrent communicating processes with stable storage on a single processor. The single computer algorithms provide optimized journaling and recovery for the single node, while the distributed algorithms would handle inter-node recovery. This is new in that distributed recovery algorithms have ignored the single computer concurrent application facet of a system.

4.4 Summary

Two areas of multiple computer fault tolerance were addressed in this chapter. The first area is that of mirrored systems. Software algorithms based on the sentry mechanism were presented that allow two or three off-the-shelf processors to appear as one highly reliable, continuously available processor. Application-transparency again means that no modification to any existing software is necessary. Synchronization is maintained through the use of data and execution synchronization points. Fault detection options like detection through synchronization and detection through data comparison are presented that can allow a system to be tailored to the performance and reliability needs of the target environment. The integration algorithms make it possible to replace a failed processor with a new one with minimal disruption to system service. Fault free overhead of such systems is calculated to be typically less than 1 to 2 percent. These figures, combined with the many advantages of a purely software approach to high availability make this approach fill a long existing void in commercial reliability offerings. The projected systems compare extremely well with hardware based solutions that are orders of magnitude more expensive.

The second area of multiple computer fault tolerance addressed is that of distributed computing. Existing algorithms that provide recovery of failed distributed applications were reviewed and two new algorithms presented. The new algorithms, the optimistic hybrid and the pessimistic hybrid, are both based on hybrid checkpointing, an approach to consistent checkpointing that provides for loose synchronization instead of the performance expensive, tight synchrony of previous algorithms. Systems where output commit is frequent are targeted by the pessimistic hybrid whereas the optimistic hybrid algorithm is best for systems with infrequent output. These algorithms both journal only message order instead of the content based journaling of previous algorithms, potentially offering improved performance. Integration of distributed recovery algorithms with the sentry mechanism is discussed and the contention is that the sentry mechanism is an ideal frame work for providing their application-transparent execution.

Chapter 5

Contributions and Future Directions

5.1 Contributions

The goal of this thesis was to define, implement and demonstrate an effective support mechanism for application-transparent fault management. A proposed infrastructure for dynamically tunable, application-transparent fault management policies, called the sentry mechanism, was defined and implemented. The sentry mechanism is based on the concept of operating system service encapsulation. It was hypothesized that encapsulation, with optional service bypass, would provide sufficient power and visibility to meet the demands of even the most complex and demanding fault management policies. The hypothesis has proven to be correct. An operating system monitoring policy was developed and with its use a software fault was discovered in 3.0/UX. The fault was diagnosed with the aid of a visualization tool that displays monitor data and an assertion policy written to prevent operating system failure in the presence of the fault. These examples served as initial proof of the concept's validity.

In order to convincingly validate the sentry concept, a single area of fault management, fault recovery, was selected as representative of highly system dependent and complex fault management areas. By proving that the sentry mechanism is sufficient for the design and implementation of practical fault tolerant techniques spanning the range of computer configurations, from single computer to mirrored systems to distributed systems, the sentry concept would be considered sufficiently demonstrated.

Prior to this work, the modern computing environment of highly interactive, concurrent, disk based applications had not been fully investigated with respect to practical software based fault tolerance. This thesis has presented the design and implementation of single computer fault tolerance with the use of journaling and checkpointing policies. Development of these policies lead to new algorithms for concurrent process recovery, disk checkpointing, and checkpoint snapshots. Problems confronting the design of a practical solution were identified and individually addressed. As a result of the combination of the journaling and checkpointing, application-transparent fault tolerance is provided to these modern applications. Hard disk drives, a common and inexpensive commodity, are shown to be an ideal and efficient stable storage medium. When the policies are in effect a single computer, concurrent application, like the pervasive X window system and its client programs, is provided fault tolerance with less than 10 MB of recovery data storage requirements and less than 5% performance overhead in the average case.

Multiple computer fault tolerance was divided into the two areas of mirrored systems and distributed systems. Algorithms ideal for sentry mechanism implementation were defined that allow off-the-shelf processors to be seamlessly connected, providing a system view of one, highly available system. These new algorithms, which all integrate well with the sentry concept, solve the issues of processor synchronization, fault detection and integration of a new processor in the event of a failure. Performance projections show less than a 1% overhead for typical applications and less than a 30 minute integration period for attaching a new node. Because these systems provide continuous availability with software, they are attractive alternatives to the much more costly hardware based systems that are prevalent today.

Finally, the applicability of the sentry concept in the distributed recovery domain has been investigated. As an extension of the work on single computer fault tolerance distributed recovery algorithms have been developed that are based on the principle of saving message order, rather than content. The new recovery algorithms, the pessimistic hybrid and the optimistic hybrid, allow applications to take loosely synchronized, yet globally consistent checkpoints. The combination of journaling only message order and the loosely synchronous checkpointing can offer performance advantages over previous algorithms.

By exploring software based, application-transparent, fault tolerance based on the sentry concept across the domains of computing environments - single computer, mirrored systems, and distributed systems, the validity of the sentry mechanism has been experimentally verified. Performance efficient solutions have been presented for all of these areas. Most significantly, an advanced prototype of the application-transparent single computer algorithms has been implemented and is fully functional, capable of recovering the largest and most complex of single computer applications with, in most cases, negligible overheads.

5.2 Future Directions

There are several orthogonal areas for future research directions. In the single computer domain, work remains to be done with respect to the issue of network based applications that cannot use distributed recovery algorithms because some nodes are not recovery participants. This is the case if a fault tolerant machine is communicating with a foreign machine that has not been modified to be aware of recovery protocols. Recovery in a network environment poses new challenges concerning practicality of disk based stable storage as well as types of recovery possible.

Further work is needed to both experimentally validate the mirrored system algorithms and overcome implementation details. Integration of the mirrored system ideas into a real network environment promises to be an exciting undertaking.

Study of distributed recovery algorithms in realistic environments is an area where little research has taken place. The increasing number of such algorithms (contributed to here) is a confusing maze of trade-offs with no real experimental data. A thorough comparison of these algorithms working under practical conditions is necessary.

Application-transparent fault detection policies based on the sentry mechanism must be investigated. Development of effective and practical fault detection would compliment the sentry fault tolerant policies as well as bring existing distributed systems closer to their theoretical assumption of fail-stop nodes.

Because many systems that require fault tolerance also have real-time execution constraints, integration of the policies with real-time system design is desirable.

Finally, massively parallel computers (MPPs) introduce new levels of communication and interconnection that existing fault tolerant techniques are not equipped to handle. Failure rates for individual nodes are high enough that when hundreds or thousands are connected together, failures occur every few hours making efficient fault tolerance solutions to the MPP environment an area of interest.

Chapter 6

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub and R. Rashid, "A new kernel foundation for UNIX development", *USENIX 86*, July 1986.

[1] D. M. Andrews, "Software fault tolerance through executable assertions", in *12th Asilomar Conf. Circuits and Syst. and Comput.*, Pacific Grove, CA., Nov. 1978, pp. 641-645.

[2] D. M. Andrews, "Using executable assertions for testing and fault tolerance", in *Proc. 9th Int. Symp. Fault-Tolerant Comput.*, June 20-22, 1979, pp. 102-105.

[3] M. J. Bach, "The design of the unix operating system," Prentice-Hall, Englewood Cliffs, NJ, 1986.

[4] J. F. Bartlett, "A nonstop kernel", in *8th Symp. on Operating Syst. Principles*, Asilomar, CA., Dec. 1981, pp. 22-29.

[5] B. Bhargava and Shu-Renn Lian, "Independent checkpointing and concurrent rollback recovery in distributed systems - an optimistic approach", in *7th Symp. Reliable Distributed Syst.*, Colombus, OH, Oct. 1988, pp. 3-12.

[6] K. P. Birman, "The process group approach to reliable distributed computing," *Comm. ACM*, vol. 36, no. 12, Dec. 1993.

[7] A. Borg, W. Blau, W. Graetcsh, F. Hermann, and W. Oberle, "Fault-tolerance under UNIX." *ACM Trans. Comput. Syst.*, vol. 7, no. 1, Feb. 1989, pp. 1-24.

[8] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, Feb. 1985, pp. 63-75.

[9] G. Chappel, *DOS internals*, Addison-Wesley, 1994.

[10] H. Custer, *Inside Windows NT*, Microsoft Press, Redmond, WA. 1993.

[11] E. N. Elnozahy and W. Zwaenepoel, "Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit", *IEEE Trans. on Comput.*, vol. 41, no. 5., May 1992, pp. 526-531.

[12] T. M. Frazier and Y. Tamir, "Application-transparent error-recovery techniques for multicomputers", in *4th Conf. on Hypercubes, Conc.Comput. and App.*, March 1989, pp. 103-108.

[13] D. Golub, R. Dean, A. Forin and R. Rashid, "Unix as an Application Program", in *USENIX Summer Conference*, Anaheim, CA, June 11-15, 1990.

[14] J. Gray and D. P. Siewiorek, "High-Availability Computer Systems", *IEEE Comput.*, September, 1991.

[15] Y. Huang and C. Kintala, "Software implemented fault tolerance: technologies and experience," in *Proc. 23rd Int. Symp. on Fault-Tolerant Comput.*, June 22-24, 1993, pp. 2-9.

[16] D. Jewitt, "Integrity S2: A Fault-Tolerant Unix Platform", in *Proc. 21st Int. Symp. Fault-Tolerant Comput.*, June 1991, pp. 512-519.

[17] D. B. Johnson and W. Zwaenepoel, "Sender-based message logging," in *Proc. 17th Int. Symp. Fault-Tolerant Comput.*, June 1987, pp. 14-19.

[18] ________. "Recovery in distributed systems using optimistic message logging and checkpointing," *J. Alg.*, vol. 11, no. 3., 1990, pp. 462-491.

[19] T. T. Juang and S. Venkatesan, "Efficient algorithms for crash recovery in distributed systems", *10th Conf. on Found. of Software Tech. and Theoretical Comput. Sci.*, 1990, pp. 17-19.

[20] T. Juang and S. Venkatesan, "Crash recovery with little overhead," in *Proc. 11th Int. Conf. Distributed Comput. Syst.*, May 1991, pp. 454-461.

[21] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. Software Eng.*, vol. SE-13, Jan. 1987, pp. 23-31.

[22] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, July 1978, pp. 558-565.

[23] T. Lehr, Z. Segall, D. Vrsalovic, E. Caplan, A. Chung, and C. Fineman, "Visualizing performance debugging", *IEEE Comput.*, Oct. 1989, pp. 38-51.

[24] K. Li, J. F. Naughton, and J. S. Plank, "Checkpointing multicomputer applications," in *Proc. 10th Symp. Reliable Distributed Syst.*, Oct. 1991, pp. 2-11.

[25] A. Mahmood, D. J. Lu, and E. J. McCluskey, "Executable assertions and flight software", *AIAA/IEEE 6th Digital Avionics Syst. Conf.*, Dec. 1984, pp. 346-351.

[26] D. Mosse, O. Gudmunsson and A. Agrawala, "The MARUTI system and its implementation," Technical Report, Department of Computer Science, University of Maryland, June 1991.

[27] L. L. Peterson, N. C. Bucholz, and R. D. Schlichting, "Preserving and using context information in interprocess communications," *ACM Trans. Comput. Syst.*, vol. 7, no. 3, Aug. 1989, pp. 217-246.

[28] M. Pietrek, *Windows internals*, Addison-Wesley, 1993.

[29] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr, and R. Sanzi, "Mach: a foundation for open systems", in *Proc. 2nd Workshop Workstation Operating Syst.*, Sept. 27-29, 1989.

[30] M. Russinovich, Z. Segall, "Open system fault management - fault tolerant Mach", CMU Research Report, CMUCDS-92-8, 1992.

[31] M. Russinovich, Z. Segall, and D. P. Siewiorek, "Application transparent fault management in fault tolerant Mach", in *Proc. 23rd Int. Symp. Fault-Tolerant Comput.*, June 22-24, 1993, pp. 10-19.

[32] R. W. Scheifler and J. Gettys, "The X-window system," *ACM Trans. on Graphics*, vol. 5, no. 2, Apr. 1986, pp. 79-109.

[33] D. Siewiorek and R. Swarz, *Reliable computer systems: design and evaluation*, Digital Press, Burlington, MA. 1992.

[34] R. E. Strom, D. F. Bacon and S. A. Yemini, "Volatile logging in n-fault tolerant distributed systems", in *Proc. 18th Int. Symp. on Fault Tolerant Comput.*, Tokyo, Japan, 1988, pp. 27-30.

[35] R. E. Strom, D. F. Bacon and S. A. Yemini, "Towards self recovering operating systems", *International Conf. Reliable Syst.*, Los Angeles, CA, April 21-23, 1975, pp. 59-71.

[36] R. E. Strom and S. A. Yemini, "Optimistic recovery in distributed systems", *ACM Trans. Comput. Syst.*, vol 3, no. 3, Aug 1985, pp. 204-226.

[37] Y. Tamir and T. Frazier, "Application-transparent process-level error recovery for multicomputers," in *22nd Hawaii Int. Conf. Syst. Sciences*, Jan. 1987.

[38] Z. Tong, R. Y. Kain, and W. T.Tsai, "A low overhead checkpointing and rollback recovery scheme for distributed systems," in *8th Symp. Reliable Distributed Syst.*, Oct. 1989, pp. 12-20.

[39] *UNIX Programmer's Manual*, USENIX Association, March 1986.

Appendix

Algorithm Pseudocode

This appendix shows more detailed C language style pseudocode for the single computer algorithms of Chapter 3.

The pseudocode algorithm for journaling and replay of the journal are shown in Listing 1 and Listing 1 respectively. The routine journal() is executed in the journal exit sentry and the routine replay() is executed in the recovery policy's entry sentry. In the code, my_pid refers to the current process' identifier. The sleep() routine places the process pid and the passed number on the a specified queue and puts the process to sleep. The wakeup() routine searches the specified queue for the entry with the indicated number and wakes up the process associated with it.

An outline of the signal replay code is shown in Listing 3. Each process has a structure in which it stores the next signal it must deliver to itself and the system call count indicating the delivery time. At each system call a count is incremented and when the count becomes equal to the one in the signal record the process delivers the signal to itself. It then reads the next signal, but skips over any sleep/wakeup pairs.

The entire disk checkpointing algorithm is outlined in high level pseudocode in Listing 1 and Listing 1 with support routines in Listing 1.The data structures and defines are incomplete, but meant to be representative. The basic utility routines of read() and write() take a file pointer as the first argument (the names in the code are indicative of which file is being accessed) and a list of data arguments that are sequentially read or written. seek() moves the file read/write pointer to the specified offset in the file and write/readdata() writes/reads a block of data from the specified file. The routine file_name_modify() is called before any modification is made to the *name* state of a file. File_data_modify() is called before the data state of a file is modified and file_stats_modify() is called before a call modifies the file stats. The function hash_table_enter() puts a file control block onto a hash queue based on the value of the curname field and hash_table_lookup() looks up the file control block that has the field curname set to the passed *name* argument. The disk checkpoint restore pseudocode is shown in and Listing 7 and Listing 7.

```
/* journal entry structure */
typedef struct {
   short        pid;   /* process-id of executor */
   int          seq;   /* sequence number of event */
} jrnrec_t;

/* global variables */
int       glob_seq;     /* global sequence number */
queue     jrnq;         /* journal queue */
int       journaled;    /* sequence of last record in journal */
int       lastproc;     /* process id of last process to
                         * write a record */

/* executed in exit sentry of journaling policy
   for sequenced events */
journal() {
   int          new_glob;
   jrnrec_t     jrnrec;

   /* increment the count */
   new_glob = glob_seq++;
   /* wait until all previous records are in journal */
   while(journaled != new_glob-1)
         sleep(jrnq,my_pid,new_glob);
   /* now write this record into the journal */
   jrnrec.pid = my_pid;
   jrnrec.seq = new_glob;
   if(lastproc == my_pid)
         /* do run-length optimization */
         overwrite_journal_record(&jrnrec);
   else {
         write_journal_record(&jrnrec);
         lastproc = my_pid;
   }
   journaled++;
   /* wakeup any process waiting to write the next record */
   wakeup(jrnq,journaled);
}
```

Listing 1. Journaling algorithm

Data structures associated with the snapshot algorithm are shown in Listing 3. The snapshot algorithm itself is shown in Listing 3 and Listing 3.

```
/* journal entry structure */
typedef struct {
    short        pid;   /* process-id of executor */
    int          seq;   /* sequence number of event */
} jrnrec_t;

/* global variables */
int       glob_seq;     /* global sequence number */
jrnrec_t  jrnrec;       /* next journal record */
queue     rcvrq;        /* recover queue */
int       curpid;       /* process that will execute next
                         * sequenced event intialized to pid
                         * in first record of journal */

/* executed in entry sentry of recovery policy for
    sequenced events */
replay(){
    int new_glob;

    /* wait until its this process' turn */
    while(curpid != my_pid)
            sleep(rcvrq,my_pid,my_pid);
    /* read next record out of journal */
    if(new_glob == jrnrec.seq-1) {
            read_journal(&jrnrec);
    }
    /* now can increment the count */
    new_glob = glob_seq++;
    if(new_glob == jrnrec.seq) {
            curpid = jrnrec.pid;
            wakeup(rcvrq,jrnrec.pid);
    }
}
```

Listing 2. Journal replay algorithm

```
/* signal entry definition */
typedef struct {
    short        signal;       /* the signal type */
    int          sysnum;       /* the system call the signal
                                * was delivered*/
} sigrec_t;

sigrec_t sigrec;        /* each process has its own sigrec structure */

/* this is called in the entry sentry of the
    recovery policy for
    EVERY system call */
make_signal() {
    sigrec_t     nxtsig;

    /* increment the system call count */
    proc->sysnum++;
    /* time to deliver a signal? */
    if(sigrec.sysnum == proc->sysnum) {
            /* process delivers signal to itself */
            send_signal(my_pid,sigrec.signal);
            /* read the next signal from the signal file */
            cc = read_signal(&sigrec);
            if(cc != 0) {
                    /* skip over all SLEEP/WAKEUP pairs */
                    while(sigrec.signal == SLEEP && cc>0) {
                            cc = read_signal(&nxtsig);
                            if(cc != 0 && sigrec.signal == WAKEUP)
                                    cc = read_signal(&sigrec);
                            else
                                    bcopy(&nxtsig,&sigrec,sizeof(sigrec_t));
                    }
            } else sigrec.sysnum = 0; /* set it to NULL */
    }
}
```

Listing 3. Signal replay algorithm

```
/* stats structure */
typedef struct {
   time_t              ctime;        /* time the file was created */
   time_t              mtime; /* time the file was modified */
   int                 size;  /* size of the file */
} statstr;

/* data structure allocated for each file that is modified */
typedef struct {
   int                 new;
   int                 gone;
   int                 fileid;
   dynamicarray        blocksbacked[];
   string              origname;
   string              curname;
   statstr             stats;
} filestr;

/* this file descriptor is shared by the application */
FILE      *filecatalog;
int       numfiles;     /* counter used to assign identifiers to
                         * files */

/********* support routines ************/
/* allocate a new file control block */
new_filecb(char *name,statstr *stats) {
   filestr      *filecb;

   numfiles++;
   filecb = alloc(sizeof(struct filestr));
   filecb->fileid = numfiles;
   filecb->origname = name;
   filecb->curname = name;
   /* if its a new file name is NULL */
   if(name == NULL) filecb->new = TRUE;
   filecb->stats = stats;
   write(filecatalog,filecb,sizeof(filestr));
   hash_table_enter(filecb);
   return(filecb);
}

/* looks up a file's control block or allocates a new one */
file_lookup(char *name) {
   filestr      *filecb;
   statstr      *stats;

   filecb = hash_table_lookup(name);
   if(filecb == 0) {
         stats = read_stats(name);
         filecb = new_filecb(name,stats);
   }
   return(filecb);
}
```

Listing 4. Disk checkpoint algorithm support routines

```
/* this file descriptor are shared by the application */
FILE      *renamefile;

/* called when the name state of a file will be modified */
file_name_modify(char *old_name, char *new_name) {
    filestr      filecb;

    filecb = file_lookup(old_name);
    /* is it a delete operation? */
    if(new_name == NULL && !filecb.new) {
          /* backup file and get the backup name */
          new_name = move_to_backup(old_name);
    }
    /* delete old position in hash table */
    hash_table_remove(filecb);
    /* is it a new file? */
    if(old_name == NULL) old_name = "new";
    /* deleting a new file? */
    if(new_name == NULL && filecb->new) new_name = "gone";
    filecb->curname = new_name;
    write(renamefile,filecb->fileid,old_name,new_name);
    hash_table_enter(filecb);
}

/* called when the stats part of a file will be modified */
file_stats_modify(char *name) {
    (void) file_lookup(name);
}
```

Listing 5. Disk checkpoint algorithm fault-free execution(a)

```
/* blockfile file descriptor is shared by the application */
FILE      *blockfile;

/* called when the data state of a file is going to be modified.
   The modification affects the file starting at pos and is length
   in bytes */
file_data_modify(FILE *file,char *name,int pos,int length) {
   int    i,startblock,numblocks,endblock;
   int    startread,readlen,bit,byte,mask;

   filecb = file_lookup(name);
   /* only backup files that existed at the ckpt */
   if(!filecb->new) {
         startblock = pos/BLOCKSIZE:
         numblocks = length/BLOCKSIZE;
         /* don't worry about blocks past the end of the
          * original file */
         endblock = filecb->stats.size/BLOCKSIZE;
         startread = 0;
         readlen = 0;
         /* collect the largest contiguous unbacked data blocks */
         for(i = startblock;i <= startblock+numblocks;i++) {
               /* calculate the bit representing the block */
               byte = i/8;
               bit = 1 - byte*8;
               mask = 1 << bit;
               /* is the block backed alread? */
               if(!(filecb->blocksbacked[byte] & mask)) {
                     filecb->blocksbacked[bytes] |= mask;
                     if(!startread) startread = i*BLOCKSIZE;
                     if(i == endblock)
                           readlen += filecb->size - i*BLOCKSIZE;
                     else
                           readlen += BLOCKSIZE;
               } else if(readlen) {
                     /* read the original data and back it */
                     seek(file,startread);
                     read_data(file,data,readlen);
                     write(blockfile,filecb->fileid,
                           startread,readlen);
                     write_data(blockfile,data,readlen);
                     startread = 0;
                     readlen = 0;
               }
         }
   }
}
```

Listing 6. Disk checkpoint algorithm fault-free execution (b)

```
/* table used for fileindex->filecb translation */
filestr   *filecbtable[NUMFILES];

/* number of files in catalog */
int        numfiles;

/* rename phase is second to move all files back
   to their original positions */
rename_phase() {
   filestr       *filecb;
   int           pos;

   /* start at end of file and read backwards */
   pos = sizeof_file(renamefile);
   /* backup one entry */
   pos = pos-2*sizeof(rename_record);
   while(pos) {
          read(renamefile,fileid,origname,newname);
          filecb = filecbtable[fileid];
          if(filecb)
                 if(filecb->gone) continue;
          /* handle special cases */
          /* is it a newfile that was deleted? */
          if(newname == "gone")
                 filecb->gone = TRUE;
          else {
                 /* its a new file that needs to be deleted */
                 if(oldname == "new")
                        delete(newname);
                 else {
                        /* see if it was a file that was backed up */
                        if(newname == "/back/*") {
                               /* '*' is wildcard */
                               copyfile(newname,oldname);
                        else {
                               /* standard case -
                                do inverse of rename */
                               renamefile(newname,oldname);
                        }
                 }
          }
   }
}
```

Listing 7. Disk checkpoint restore algorithm(a)

```
/* this it the first step because it sets up the
   filecb data structures with the file identifiers */
read_file_catalog() {
   filecb = alloc(sizeof(struct filestr))
   while(read(filecatalog,filecb,sizeof(struct filestr)) {
          hash_queue_enter(filecb);
          filecbtable[numfiles++] = filecb;
   }
}

/* now, the data portion of the files are restored */
block_restore_phase() {
   int cc,fileindex,pos,len;

   while(cc = read(blockfile,&fileindex,&pos,&len)) {
          read_data(blockfile,data,len);
          seek(filecbtable[fileindex]->curname,pos);
          write_data(filecbtable[fileindex]->curname,data,len);
   }
}

/* finally, the file stats are restored */
stats_phase() {
   int i;
   statstr      stats;

   for(i=0;i<numfiles;i++)
          write_stats(filecbtable[i].curname,filecbtable[i].stats);
}

/* main recovery routine called at the beginning of a recovery */
disk_restore() {
   read_file_catalog();
   rename_phase();
   block_restore_phase();
   stats_phase();
   /* free all data structs -they are recreated dynamically
    during the recovery execution */
   free_filecbtable();
}
```

Listing 8. Disk checkpoint restore algorithm(b)

```
/* the process list data structure is maintained by the checkpointing
   policy and it reflects the process that are currently part of
   the application */
typedef struct applist_str {
   /* the ckpted flag is initially zero */
   int                        ckpted;
   /* set by sentry indicating process is waiting in syscall entry */
   int                        inentry;
   struct proc                *proc;
   struct applist_str         *next;
} applist;

/* procs in the application */
applist   approcs;
```

Listing 9. Snapshot data structures

```
/* snapshot algorithm */
snapshot() {
   int           numleft;
   applist       *curproc;

   /* first, freeze all procs */
   numleft=0;
   curproc = appprocs;
   /* tell procs to wait in system call entry */
   set_stop_in_entry_flag();
   while(curproc) {
          process_suspend(curproc->proc);
          curproc->stopped = TRUE;
          numleft++;
          curproc = curproc->next;
   }
   /* see next figure for continuation */
```

Listing 10. Snapshot algorithm (a)

```
/* routine continued from previous figure */
    while(numleft) {
            curproc = appprocs;
            while(curproc) {
                    if(!(curproc->ckpted == TRUE)) {
                            if(!curproc->stopped) {
                                    process_suspend(curproc->proc);
                                    curproc->stopped = TRUE;
                            }
                            /* in state 1? */
                            if(in_own_address_space(curproc->proc)) {
                                    checkpoint_process(curproc->proc);
                                    curproc->ckpted = TRUE;
                                    numleft--;
                            }
                            /* in 4 or in call entry? */
                            if(curproc->inentry ||
                                    in_semi_sleep(curproc->proc)) {
                                    /* if proc in system call, regs need to
                                     * fixed for restart
                                     */
                                    fix_registers(curproc->proc);
                                    checkpoint_process(curproc->proc);
                                    curproc->ckpted = TRUE;
                                    numleft--;
                            }
                            /* in state 2 or 3 (not checkpointable) */
                            if(curproc->ckpted != TRUE) {
                                    /* process need to continue a little */
                                    curproc->proc->stopped = FALSE;
                                    process_resume(curproc->proc);
                            }
                    }
                    curproc = curproc->next;
            }
            /* let process run for awhile before checking them again */
            if(numleft) delay();
    }
    curproc = appprocs;
    while(curproc) {
            process_resume(curproc->proc);
            curproc->stopped = FALSE;
            curproc = curproc->next;
    }
}
```

Listing 11. Snapshot algorithm (b)